9.95

UNDERSTANDING MICROCOMPUTERS AND SMALL COMPUTER SYSTEMS

AUTHOR:

Nat Wadsworth

© Copyright 1977
Scelbi Computer Consulting, Inc.
1322 Rear - Boston Post Road
Milford, CT 06460

- ALL RIGHTS RESERVED -

IMPORTANT NOTICE

No part of this publication may be reproduced, transmitted, stored in a retrieval system, or otherwise duplicated in any form or by any means electronic, mechanical, photocopying, recording, or otherwise, without the prior express written consent of the copyright owner.

The information in this manual has been carefully reviewed, and is believed to be entirely reliable. However, no responsibility is assumed for inaccuracies or for the success or failure of various applications to which the information contained herein might be applied.

ACKNOWLEDGEMENT

The author wishes to thank the following members of the staff at Scelbi Computer Consulting, Inc., for their assistance in the preparation of this book.

Richard Hecht
Raymond Edwards

Printing: 9 8 7 6 5 4 3 2

Printed in the U.S.A.

DEDICATION

This book is dedicated to Theresa, and to medical doctors, and to all the rest who really know what dedication is all about.

UNDERSTANDING MICROCOMPUTERS
AND
SMALL COMPUTER SYSTEMS

****
TABLE OF CONTENTS
****

Chapter ONE. . . . . .The Basic Operation of a Microcomputer

Chapter TWO . . . . . Instructions for Microcomputers

Chapter THREE . . . . Programming - At the Machine Language Level

Chapter FOUR . . . . . .Programming - Using a Higher Level Language

Chapter FIVE . . . . . .Input/Output Devices

Chapter SIX . . . . . System Considerations

Appendix A. . . . . Combined Glossary and Index

INTRODUCTION

The digital computer was first developed as a practical electronic operating device some thirty years ago. At that time a simple computing system capable of performing rather straightforward mathematical operations for use in government, academic and business applications often cost millions of dollars. The early computers took up quite a bit of space too. They often completely filled entire rooms. They were fussy when it came to performing. First, they generally had to be kept at just the right temperature in order to work properly, which required the use of expensive air conditioners for the machines - at a time when air conditioning was considered very much a luxury. Second, even when pampered royally, they were quite prone to operational failure. It has been reported that some of the first computers had mean-time-between-failure periods of just a few hours! Hence, they required constant maintenance and repair.

About ten years ago a new breed of computer became commercially available. It was much smaller in size, lower in cost, and much more reliable than the earlier machines. It could go weeks and even months without failure. Its cost was in the tens of thousands of dollars instead of millions. It could fit comfortably in a cabinet and was not quite so fussy about room temperature. These smaller computers were termed "minicomputers." They opened the door to the use of digital computers by a host of institutions and organization that could not previously afford such equipment. The use of computers started multiplying rapidly.

Just a few years ago it became technically possible to construct computer systems - having the same kinds of capabilities as those costing a million or more dollars just thirty years ago - for less than a thousand dollars! Furthermore, these new marvels can be constructed so that they occupy less space than that taken up by a typical briefcase. They can be placed for use on the top of a desk. They are so reliable that mean-time-between-failure periods are measured in terms of thousands of operating hours or years of use. These computers are built around a technology based on the micro-miniaturization of electronic circuits. They are aptly referred to as "microcomputers."

The advent of "microcomputers" has caused an explosion in the use and application of computer technology. It is believed by many to signal the beginning of a new technological period which will have a profound impact on humankind's development. Why all the excitement and fuss about the new technology? Because the development of the microcomputer has opened the door to the application of computer techniques by individuals. The creation of the microcomputer has, for the first time, made it possible to build digital computers that can be marketed at a price individuals can afford. It has made it possible to build a computer small enough to be kept unobtrusively in a home - or even to be carried around from home, to work, to school! For the first time it has become possible for individuals to start reaping the benefits of having a powerful computing machine available at their command!

Unfortunately, a large proportion of society was caught totally unprepared to capitalize on the arrival of computer systems capable of performing at the personal level. The vast majority of people have never been educated as to how a computer works. They do not know what it can do. They do not know how to make it do things for themselves. As a matter of fact, the technology allowing people to have their own computers arrived at a time when a number of people were confused and misinformed (even frightened) about the general capabilities of computers. This may well have been because for a number of years institutions had been using the machines to study, sort, classify and in some instances, it seemed, to dehumanize people. Then too, businesses had been using computers for years - to make sure people paid their bills (sometimes with blatant disregard to the fact that they had already been paid)! Computers, it was often charged, were used to put people out of work. Even science fiction writers seemed to harp on stories of computers going berserk and taking over or annihilating humankind. Somehow the idea has been fostered over the years that computers have a mind of their own - and a somewhat sinister one at that! Such is really not the case. But, a lot of people have never had the opportunity to learn how a computer really operates, and how it can be instructed to perform valuable services for individuals. They have not had the opportunity to learn the fundamental concepts behind its operation that make it such an exciting and potentially useful machine for them personally!

The purpose of this book is to take the mystery out of the operation and practical application of microcomputers. To describe how they work, to indicate how their operation is controlled, to illustrate how one tells them what to do, to describe the practical accessories to which they are typically connected; and to give a little advise to the person planning on acquiring a microcomputer system.

Though the subject matter is somewhat technical in nature, an attempt has been made to present the material in a light and easy to read style. The material can first be approached as a straight text to gain an overview of the subject matter. Later, specific sections may be referred to again or studied in detail as desired. A combined glossary and index in the back of the publication serves as a quick reference to the technical vocabulary presented.

It is hoped that the material contained herein will enlighten the reader, dispell any myths about the subject one may have acquired in the past, and help put microcomputers in the perspective in which they belong. They are valuable tools for all humankind to use and enjoy.

THE BASIC OPERATION OF A MICROCOMPUTER

There have been numerous examples put forth over the years to illustrate the basic scheme behind the operation of computers. The scheme is deceptively simple and incredibly powerful. The power comes from the speed with which the machines can perform the simple operations. The fundamental concept of the computer is that it is a machine that is capable of doing two fundamental operations at very high speed. First, it is able to obtain a piece of information from a storage area and perform a function as directed by the information it obtains; secondly, based on its current status, it is able to ascertain where to obtain the next piece of information that will give it further directions. This fundamental concept is the key to the operation of all digital computers, and while it is a simple concept, it can be built upon to arrive at all the complex operations computers of today can perform. How this is done is what this book is about.

One of the best analogies for describing a computer's basic operations is to consider a bank of boxes, similar to a bank of post office boxes. A piece of paper containing directions can be placed in each box. A person is directed to go to the bank of boxes, and, after starting at a given place, to open each box, withdraw the piece of paper, and follow the directions thereon. The boxes are labeled in an orderly fashion, and the person is also told that unless a piece of paper in a box directs otherwise, when the person is finished performing the task directed they are to replace the paper in the box and proceed to open the next box. Note, however, that a piece of paper may give directions to alter the sequence in which the person is to open boxes.

Figure 1 shows a picture of a set of such boxes. Each box is labeled for identification.

To present a view of a computer's operation, assume a person has been told to start at box A1 and to follow the directions contained on the pieces of paper in the boxes until a piece of paper containing the direction "STOP" is found in one of the boxes. In this example, the

person finds the following instructions:

In box A1 is the message: "Take the mathematical value of 1 and write it down on a scratch pad."

Since the instruction in box A1 only pertained to some function that the person was to perform, and did not direct the person to go to some specific box, then the person will simply go on to the next box in the row. Box A2 contains the information:

"Add the number 2 to any value already present on your scratch pad."

The person would at this point perform an addition, and obtain a total accumulated value on the pad of scratch paper. The accumulated value would be 3. Since there are no other directions in box A2, the operator would continue on to open box A3 which has the following message:

"Place any accumulated mathematical value you have on your scratch pad into box H8."

Thus the person would tear the current sheet off the scratch pad and place it, containing the value 3, into box H8. Note though, that while the person was directed to place the accumulated value on the scratch pad into box H8, the person was not directed to alter the sequence in which to obtain new instructions. So the person would proceed to open box A4, which contains the directive:

"Take the mathematical value of 6 and place it on your scratch pad."

Going to box A5 the person finds:

"Add 3 to the present value on your scratch pad."

The operator adds the value 6 from the previous box to the number 3, noting the calculation on the scratch pad, and proceeds to open box A6:

A1	A2	A3	A4	A5	A6	A7	A8
B1	B2	B3	B4	B5	B6	B7	B8
C1	C2	C3	C4	C5	C6	C7	C8
D1	D2	D3	D4	D5	D6	D7	D8
E1	E2	E3	E4	E5	E6	E7	E8
F1	F2	F3	F4	F5	F6	F7	F8
G1	G2	G3	G4	G5	G6	G7	G8
H1	H2	H3	H4	H5	H6	H7	H8

FIGURE 1

"Place any accumulated value you have on your scratch pad into box H7."

The person thus would put the value '9' on a piece of paper (from the scratch pad), place it in the designated box, and then proceed to open box A7:

"Get the value presently stored in box H8 and save the value on your scratch pad."

This is a simple operation. The person would proceed to open up box A8:

"Fetch the value in box H7. Subtract the value on your scratch pad from the value found in box H7. Leave the result on your scratch pad."

When the operator had performed this operation, row A would have been finished. The operator would continue obtaining instructions by going to the B row and opening box B1:

"If the present value on your scratch pad is not zero, go to box B3."

At this time, if the person checked the scratch pad it would be found that the value on the pad was indeed non-zero. This is because the last calculation performed was to subtract the value in box H8 from the value in box H7. In this example the result would be:

$$9 - 3 = 6$$

Therefore, the directions in box B1 for this particular case tell the operator to jump over box B2 and go to box B3. For the sake of completeness, however, box B2 would contain an instruction. If the value on the scratch pad had been zero, the operator would not have jumped over box B2. The following would have been found inside box B2:

"The values in box H7 and H8 are of equal value. STOP!!"

However, for the values used in this example, the person would have jumped over box B2 to box B3 where the following directive would have been found:

"If the present value on your scratch pad is a negative number, jump to box B5."

Since that would not be the case the person would not jump to box B5, but would simply continue on to open box B4 which contains:

"The value in box H7 is larger than the value in box H8. STOP!!"

At this point the person would have completed the instruction sequence for this example. It should be noted, however, that box B5 contained the message:

"The value in box H7 is smaller than the value in box H8. STOP!!"

This little example of a person opening up boxes and following the directions contained in each one is very similar to the concept used by a computer. Note that each instruction is very short and specific. Also note that the combination of all the instructions in the example will result in the person being directed to solve the problem:

Is 1 + X greater than, less than, or equal to: 6 + Y?

For, the reader may note, if the data words contained in boxes A2 and A5 in the example were changed, the sequence of instructions would still result in the person being told to STOP at the box that contained the correct answer. The reader can verify this by simply assuming that different numbers than those used in the example are in boxes A2 and A5. Going through the instruction sequence until told to STOP will yield the correct answer for the numbers chosen.

The example illustrates how a carefully planned set of directions, arranged such that they are performed in a precise sequence, can be used to solve a problem even though the variables (data) in the problem may vary. Such a set of instructions is often termed an algorithm by those in the microcomputer field. The example solved a mathematical problem using an algorithm. However, the reader will find that algorithms can be devised to solve many problems on a computer that are not strictly mathematical!

Any person learning a new skill must, of necessity, learn the vocab-

ulary of the field in order to proceed to any great extent. One might think that it would be easier if everything was written in plain, everyday words, but the truth of the matter is that specialized vocabularies do serve several useful functions. For one thing, they can greatly shorten the time that it takes to communicate ideas or concepts. In today's fast-moving world that is of significance in itself. In addition, the limitations of the English language often result in a given word having a special meaning when it is used in the context of a particular subject. One must know the new meaning when it is used in such a manner. Throughout this book the meanings of various words used by those in the microcomputer field will be pointed out. Fortunately, much of the microcomputer field vocabulary is very logically named. This is probably due to the fact that computers are, of necessity, extremely dependent on logic, and hence, many persons who helped create the field (and by that fact were rather logically oriented themselves) seem to have had the sense to have named many of the parts and systems of computers and computer programs in a logical manner.

On the next few pages are two diagrams (figures 2A and 2B) which shall be used to demonstrate the analogy between a person following instructions obtained from a group of mail boxes and the fundamental organization and operation of a real microcomputer.

Figure 2A shows a group of post office boxes, a person who is able to fetch and return the instructions or data from and to the boxes, and a scratch pad on which the person can make calculations when so directed.

Figure 2B shows three interconnected boxes which represent a block diagram of a basic microcomputer. The upper-most portion of the block diagram is labeled the MEMORY. The middle portion is labeled the CENTRAL PROCESSOR UNIT or CPU for short. The lower part of the diagram depicts an ACCUMULATOR.

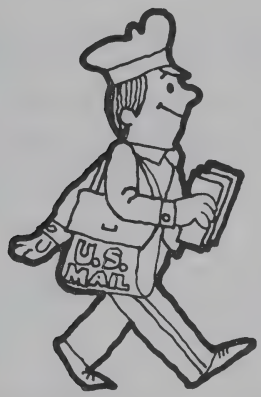
The correlation between the two pictures is extremely simple. The post office boxes correspond to the memory portion of a real microcomputer. The MEMORY is a storage area. It is a place where instructions and data can be stored for long lengths of time. The memory can be accessed. Instructions and/or data can be taken out of mem-

POST OFFICE BOXES

A1	A2	A3	A4	A5	A6	A7	A8
B1	B2	B3	B4	B5	B6	B7	B8
C1	C2	C3	C4	C5	C6	C7	C8
D1	D2	D3	D4	D5	D6	D7	D8
E1	E2	E3	E4	E5	E6	E7	E8
F1	F2	F3	F4	F5	F6	F7	F8
G1	G2	G3	G4	G5	G6	G7	G8
H1	H2	H3	H4	H5	H6	H7	H8

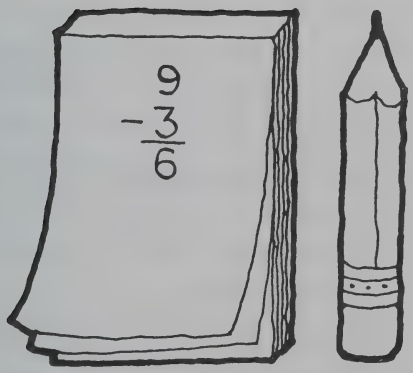
= MEMORY

PERSON



= CENTRAL PROCESSING UNIT

PAD & PENCIL



= ACCUMULATOR

FIGURE 2A

```
*****************************
*                           *
*                           *
*                           *
*                           *
*          MEMORY           *
*                           *
*                           *
*                           *
*                           *
*****************************
          .         .
          .         .
          .         .
          .         .
          .         .
*****************************
*                           *
*                           *
*          CENTRAL          *
*         PROCESSOR         *
*           UNIT            *
*          (CPU)            *
*                           *
*                           *
*****************************
          .         .
          .         .
          .         .
          .         .
          .         .
*****************************
*                           *
*                           *
*       ACCUMULATOR         *
*                           *
*                           *
*****************************
```

FIGURE 2B

BLOCK DIAGRAM OF THE FUNDAMENTAL COMPONENTS OF A MICROCOMPUTER

ory, operated on, and replaced. New data can be put into the memory. A memory that can be read from as well as written into is called a read and write memory. A read and write memory is often referred to as a RAM for abbreviation. Many times it is feasible to have a memory that is only read from. A memory that is never written into, but is only used to read from, is termed a "read-only-memory," and is abbreviated as a ROM. For the present discussion the term "memory" will refer to read and write memory (RAM). The utilization of read only memories, as a general rule, requires a more sophisticated central processor unit and more sophisticated programming techniques than those being illustrated in the present discussion. Hence, their use will not be discussed at this time.

The figure of a person in picture 2A corresponds to the CENTRAL PROCESSOR UNIT in figure 2B; The CENTRAL PROCESSOR UNIT in a computer is the section that controls the overall operation of the machine. The CPU can receive (fetch) instructions or data from the memory. It is able to interpret the instructions it fetches from the memory. It is also able to perform various types of mathematical operations. It can also return information to the memory to, for instance, make deposits of data into the memory. The CPU contains control sections that enable it to sequentially access the next location in memory when it has finished performing an operation. Or, if it is directed to do so, to access the memory at a specified location, or to jump to a new area in memory from which to continue fetching instructions.

The pad of paper and pencil in figure 2A corresponds to the block titled ACCUMULATOR in picture 2B. The ACCUMULATOR is a temporary register or manipulating area which is used by the CPU when it is performing operations, such as adding two numbers. One number, or piece of information, can be temporarily held in it while the central processor unit goes on to obtain additional instructions or data from memory. It is an electronic scratch pad for the CPU.

The three fundamental units: the memory, central processor unit, and the accumulator are at the heart of every digital computer system. Of course, there are other parts which will be added in and explained later, but these fundamental portions can be used to explain the basic operation of a digital microcomputer which is the pur-

pose of this chapter.

The reader should learn the names of the basic parts of a microcomputer as they are presented. Note how easy it is to remember the portions that have been shown. The remembering element is the memory. The portion that does the work or processing is simply termed the central processor unit, and the part that is used to accumulate information temporarily is aptly called the accumulator!

The reader now has a fundamental view of the concept behind a computer's operation, and an understanding of the machine's most basic organization. It is simply a machine that can fetch information from a memory, interpret the information as an instruction or data, perform a very small operation, and continue on to determine the next operation that is to be performed. Each operation it is capable of doing is very tiny by itself. But, when the many operations of a typical program are performed in sequence, the solutions to very complex problems can be obtained. It is important to remember that the microcomputer can perform each little operation in just a few millionths of a second! Thus, a program that might seem very large to a person, say one with many thousands of individual instructions, would only take a digital microcomputer a few thousandths of a second to perform! The speed with which the computer can execute individual instructions is what gives the microcomputer its fantastic capability.

It is now time to start delving into the actual physical manner in which a microcomputer operates. How can a machine be constructed so that it is able to perform the processes of the central processor unit? While it will require a number of pages of text to explain the procedure, it is not nearly as difficult to understand as many people might suspect. The complexity of a computer when first viewed by a person is caused by the fact that it appears to consist of many hundreds of parts. It becomes much simpler when one understands that the hundreds of parts are really made up from a small number of similar parts and these are carefully organized into several main operating sections. The reader is already familiar with the most fundamental portions.

As fantastic as it may sound at first, a digital microcomputer can

be thought of as really nothing more than a highly organized collection of on or off switches! Yes, computers are constructed from electronic devices that can only assume one of two possible states! The electronic switches can be constructed in a variety of ways. For instance, the switch can be made so that the voltage at a given point is either high or low, or current through a device is either flowing or not flowing; or flowing in one direction, and then the other direction. But, regardless of how the electronic switch is constructed, its status can always be represented as being either on or off. This on or off status can be mathematically symbolized most suitably by a mathematical system based on binary notation.

Some people tend to think that computers are very difficult to understand because they have heard of "strange" types of mathematics that are often referred to in conjunction with computers. In actuality, most of the mathematical operations dealt with in computer technology are much easier to understand and deal with than the decimal system with which the average person is familiar. In the decimal numbering system a person must learn ten different symbols. In order to manipulate those symbols, they must memorize a lot of information. For instance, look at how students are taught to multiply. The learning process actually involves them having to memorize a rather large number of facts. Because of the way it is typically taught, most students never realize how much work they have to go through just to learn the multiplication tables! The teacher does not stand up and say, "O.K., now you are going to memorize about 100 facts." Instead, over a period of a few weeks or so, the student is urged to memorize the 100 or so facts - a few at a time. The student must learn the value of each digit multiplied by all the other digits in the decimal numbering system. The decimal numbering system is far more complicated for the beginner than learning the binary numbering system. The binary numbering system is the one utilized by computers at their most basic functioning level. The reason the computer uses the binary system is because it is the simplest system around. Hence, it is the easiest one around which to construct a computing machine!

Readers know the word BINARY indicates "two." Computers are built up of electronic switches that can only have two possible states. The switches are binary devices. The status of the switches can be re-

presented mathematically utilizing the binary numbering system. The binary numbering system only has two digits in it! They are zero (0) and one (1). A switch can thus be mathematically symbolized, for instance, by a zero when it is off, and one when it is on. The opposite relationship could also be established, a one representing a switch being off and a zero used to represent a switch as on. It would make no difference mathematically which convention was used as long as one was consistent. For the purposes of the present discussion, the reader can assume that the first convention (switch off = 0, switch on = 1) will be used.

It should be immediately apparent that working with a numbering system based on only two integers will be a lot easier than working with one having ten integer symbols. In fact, most problems for people learning the binary system come about because they tend to forget how simple it is, and they keep going towards a decimal solution out of habit when they are working with the binary system. For instance, when one starts to add binary numbers, as soon as the value '1' is exceeded, a carry to the next column must be made. The value of the addition "1 + 1" in the binary system is 10. It is not 2! There is no such integer as '2' in the binary numbering system. However, when a person who has worked with the decimal system for years first starts working with the binary system, old decimal habits tend to get in the way. The reader will have to beware!

To formally introduce the binary mathematical system, one can start by stating that it uses two integers (zero (0) and one (1)), and no others. A binary number has a value determined by the value of the integers that make up the number and the position of the digits.

In the decimal numbering system, the reader is familiar with the location of a digit having a weighted value as follows: a three digit number has a value determined by the unit value of the digit in the right-most column, plus the value of the digit to the left of it multiplied by ten, plus the value of the third digit multiplied by one hundred, as illustrated in the following example.

The decimal number 345

Is equal to:

	5 Units	=	5
Plus (+)	4 times 10	=	40
Plus (+)	3 times 100	=	300

In other words, after the right-most column (which has the value of the digit), each column to the left is given a weighting factor which increases as a power of the total number of digits utilized by the numbering system. Note that in the above example the 4 representing 40 units is equal to 4 times the number of integer symbols in the decimal system (10) because it is located in the second column from the right. The number 3 representing 300 units is equal to 3 times the number of integer symbols in the decimal system squared because it is located in the third column from the right. For mathematicians, this relationship of the weighted value of the digits based on their position can be described in mathematical shorthand as follows:

If the number of different integer symbols in the numbering system is U (for the decimal system U=10)

and the column whose weighted value is to be determined is column number M (starting with the right-most column and counting to the left)

and any digit is represented by the symbol X

Then the weighted value of a digit in column M is expressed as:

X times U raised to the power (M-1) or X*((U↑(M-1))

Note: An asterisk (*) will be used here to denote multiplication. An up-arrow (↑) to signify raising a number to a power.

The reader can easily verify that the above formula applies to the decimal numbering system. However, the above formula is a general formula that can be used to determine the weighted positional value of any numbering system. It will be used to determine the weighted positional values of numbers in the binary numbering system.

In the binary numbering system there are just two different integer symbols (0 and 1). Thus U in the previous formula is equal to 2. For illustrative purposes, assume the following binary number is to be analyzed:

1 0 1

and it is desired to determine its value in terms of decimal numbers. (Remember its binary value is just 101.) Using the previous formula for the digit in the right-most column M is equal to 1, thus (M-1) is equal to 0, and with X = 1:

$$\text{Weighted value} = X*((U\uparrow(M-1)) = 1*((2\uparrow(0)) = 1*1 = 1$$

(Remember that any number raised to the zero power is equal to 1.) Going on to the next digit it can be seen that the weighted value is simply 0! Finally, the digit in the third column from the right has the weighted value because of its position:

$$\text{Weighted value} = 1*((2\uparrow(3-1)) = 1*((2\uparrow 2)) = 1*4 = 4$$

Then, by adding up the sum of the weighted values (similar to that done for the decimal example earlier) one can see that the decimal equivalent of 101 binary is 5:

The binary number 101

is equal to:

	1 units	=	1
Plus (+)	0 times 2	=	0
Plus (+)	1 times 4	=	4

And thus 101 in the binary numbering system is the same as 5 in the decimal numbering system.

There will be more to learn about the binary numbering system. However, the brief information given will be enough to continue on with the discussion that this section is primarily concerned with: the basic operation of a microcomputer. The reader is now aware that a computer is composed of numerous electronic switches and knows that one can use a mathematical shorthand to represent the status of the switches (whether they are on or off). The reader is also aware of the fundamental concept behind a computer's operation. Thus, it is now possible to proceed to show how electronic switches can be arranged to build a functional computer. That is, how the electronic switches can be arranged and interconnected in a fashion that will allow a machine to fetch a piece of information from a memory section; decode that information so as to determine an instruction; and finally determine where to obtain the next instruction or obtain additional data.

To begin this part of the discussion, it may be beneficial for the reader to picture a group of cells (similar to the post office boxes shown earlier) arranged in orderly rows as shown in figure 3. This time, instead of each cell holding a complete instruction, it may be understood that each cell only represents part of an instruction, and that it takes a whole row of cells to make up an instruction. Furthermore, each cell may only contain the mathematical symbol for a one (1) or a zero (0). In other words, its contents represent the status of an electronic switch.

At this time, a few more computer technology definitions will be presented. In figure 3, each box containing a binary 1 or 0 represents what is called a BIT of information. While each cell may only contain one piece of information at a time, a cell can actually represent one of two possible states. It either contains a zero or a one. If one starts assigning positional values to cells in a row, it can be seen that the total number of possible states in one row will increase rapidly. For instance, two cells in a row can represent up to four states of information. This is because two cells side-by-side, containing either a zero or a one in each cell can have one of the following four states

```
                 ..................................
WORD NO. 1       * 1 * 0 *1  * 0 * 1 * 0 * 1 * 0 *
                 ..................................
WORD NO. 2       * 0 * 1 * 0 * 1 * 0 * 1 * 0 * 1 *
                 ..................................
WORD NO. 3       * 1 * 1 * 0 * 0 * 1 * 1 * 0 * 0 *
                 ..................................
WORD NO. 4       * 0 * 0 * 1 * 1 * 0 * 0 * 1 * 1 *
                 ..................................
WORD NO. 5       * 1 * 1 * 1 * 1 * 0 * 0 * 0 * 0 *
                 ..................................
WORD NO. 6       * 0 * 0 * 0 * 0 * 1 * 1 * 1 * 1 *
                 ..................................
WORD NO. 7       * 1 * 1 * 1 * 1 * 1 * 1 * 1 * 1 *
                 ..................................
WORD NO. 8       * 0 * 0 * 0 * 0 * 0 * 0 * 0 * 0 *
                 ..................................
```

FIGURE 3

at a particular moment in time: 10, 01, 11, or 00. Three cells in a row can represent up to eight states of information as the possible states of three cells side-by-side are: 000, 001, 010, 011, 100, 101, 110, 111. In fact, when each cell can represent a binary number, the total number of states of information that a row of N cells can represent is: 2 to the n'th power (2↑N). Thus, a row of eight binary cells can represent 2 to the eighth (256) states of information. That is, the combination of the eight cells can be filled with zeros and ones in 256 different patterns.

A group (row) of cells in a computer's memory is often referred to as a WORD. A WORD in a computer's memory is a fixed-sized group of cells that are accessed or manipulated during one operational cycle of the central processing unit. The CPU will effectively handle all the cells in a word in memory simultaneously whenever it processes information in the memory. Digital computers can have varying word lengths depending on how they are engineered. Most microcomputers

have a memory word size consisting of eight cells. The number of cells in a word, and the number of words in a computer's memory have a lot to do with the machine's overall capability. Most microcomputers allow the memory size of the machine to be easily expanded to increase overall computing power. Typical systems may have from 256 to 65,536 words in memory. Since each word actually contains eight cells, the number of cells in a typical microcomputer is eight times the number of words in the memory. Thus, typical systems have from 2,048 to 524,288 cells in memory. This makes it possible to store fairly large amounts of information in the computer's memory at one time.

The astute reader may have already figured out a very special reason for grouping cells into words in memory. It was pointed out earlier that a row of eight cells could represent up to 256 different patterns. Now, if each possible pattern could be decoded by electronic means so that a particular pattern could specify a precise instruction for the central processor unit, then a large group of instructions would be available for use by the machine. That is exactly the concept used in a digital computer. Patterns of ones and zeros, organized into a computer word, are stored in memory. The CPU is able to examine a word in memory and decode the pattern contained therein to determine the precise operation that it is to perform. Most microcomputers, because of technical reasons, do not decode every one of the possible 256 patterns that can be held in a row of eight cells as an instruction. Typically, about 200 of the different patterns of ones and zeros which can be represented in a eight cell memory word are used. Each pattern that represents an instruction can be decoded by the CPU, and will cause the CPU to perform a specific function.

There is another ingredient necessary for making the machine automatic in operation. That is that the CPU must know where to obtain the next instruction in memory after it completes an operation. That function is greatly aided by having the memory cells grouped as words. The reader should note that in Figure 3 each group of cells representing a word was labeled as: Word No. 1, Word No. 2, etc. There is a special portion of the central processor unit that is used to control where the next word containing an instruction in memory is located. This special part is commonly referred to as

the PROGRAM COUNTER. One reason it was given the name PROGRAM COUNTER is because most of the time all it does is count! It counts memory words! Each word in memory is considered to have an ADDRESS. In Figure 3 each word was given an ADDRESS by assigning each word a number. Word No. 1 has an address of 1, Word No. 2 has an address of 2, etc. The program counter portion of the CPU keeps tabs on where the CPU should obtain the next instruction by maintaining an address of the word in memory that is to be processed! About 90 percent of the time all the program counter does is increment the value it has each time the CPU finishes doing an operation. Thus, if the computer were to start executing a simple program that began by its performing the instruction contained in Word No. 1 in memory, the very process of having the machine start the program at that location in memory would cause the program counter to assume a value of one. As soon as the CPU had performed the function, the program counter would increment its value to two. The CPU would then look at the program counter and see that its next instruction was located in Word No. 2 in memory. When the instruction in Word No. 2 had been processed, the program counter would increment its value to three. This process might continue uninterrupted until the CPU found an instruction that told it to stop operations.

A sharp reader might be starting to ask "why have a program counter if each instruction follows the next?" The answer is simply that the availability of a program counter gives the freedom of not having to always take the instruction at the next address in memory. This is because the contents of the program counter can themselves be changed when the CPU detects an instruction that directs it to do so! This enables the computer to be able to jump around to different sections in memory and, as will become apparent later, greatly increases the capability of the machine. Furthermore, typical microprocessors have a feature that allows the particular value of the program counter at any given time to be saved (on a stack). This feature allows the computer to be able to remember a whole group of memory addresses. This enables the machine to perform very complex operations referred to as subroutining. These operations will be explained in detail further on in this manual.

The program counter is actually just a group of cells in the CPU

that may contain either a binary zero or one. The binary value in the row of cells that constitute the program counter determines the address of a word in memory. Since the number of words in memory can be very large, and since the program counter must be capable of holding the address of any possible location in memory, the number of cells in a row in the program counter are larger than the number of cells in a word of memory. In typical microprocessors, the number of cells in the program counter is 16. Since 2 to the 16th power is 65,536, the program counter can present up to 65,536 different patterns. Each pattern can be used to represent the address of a word in memory. Figure 4 illustrates what the contents of the program counter would look like when it contained the address for a specific word in memory. The address the example displays is address 0, which can be considered the first word in memory. The reader should note that an address of zero can actually represent a word in memory!

```
*************************************************
*  *  *  *  *  *  *  *  |  *  *  *  *  *  *  *  *
* 0 * 0 * 0 * 0 * 0 * 0 * 0 * 0 | 0 * 0 * 0 * 0 * 0 * 0 * 0 * 0 *
*  *  *  *  *  *  *  *  |  *  *  *  *  *  *  *  *
*************************************************
```

FIGURE 4

Earlier it was stated that some instructions can actually change the value of the program counter and thus allow a program to jump to different sections in memory. However, the reader now knows that a word in memory only contains eight cells, and yet the program counter contains 16 cells. In order to change the entire contents of the program counter (by bringing in words from memory) it is necessary to use more than one memory word! This can be done if the program counter is considered to actually be two groups of cells connected together. Each group contains eight cells. In order to change the contents of the entire program counter, one whole eight

cell word could be read from a memory location and placed in the left hand group of eight cells of the program counter. Then another eight cell word could be read from memory and placed in the right hand group of eight cells.

In order to make it easier for a person working with the machine to remember addresses of words in memory, a concept referred to by computer technologists as PAGING is utilized. PAGING is the arbitrary assignment of blocks of memory words into sections that are referred to figuratively as pages. The reader should realize that the actual physical memory unit consists of all the words in memory with each word assigned a numerical address that the machine utilizes. As far as the machine is concerned, the words in memory are assigned consecutive addresses for Word No 0 on up to the highest word number contained in the memory. However, people using computers have found it easier to work with addresses by arbitrarily grouping blocks of words into "pages." Most eight-bit microprocessor manufacturers define a memory page to be a block of 256 memory words. The first memory word address in a microcomputer is generally address zero (0). Programmers typically refer to this location as Word No. 0 on Page 0. The 256th word in memory as far as the computer is concerned has an address of 255. (Note: since the address of 0 is actually assigned for the first physical word in memory, all succeeding words have an address that is one less than the physical quantity!) A programmer could refer to this word as Word No. 255 on Page 0. The 257th word in memory has an absolute address of 256 as far as the machine is concerned. A programmer could refer to that word location as being on Page No. 1 at Location 0! Similarly, the 513th word in memory, when the paging concept is used, would be Word No. 0 on Page No. 2 for a programmer, but it is simply location 512 as far as the machine is concerned.

The reader may have noted a nice coincidence in regards to the assignment of paging in typical microcomputers. Each page refers to a block of memory words that contains 256 locations (0 - 255). The reader may recall that that is exactly the number of different patterns that can be specified by a group of eight binary cells, and that there are eight binary cells in a memory word. The relationship is more than coincidental! Note that now one has devised a convenient way for a person to be able to think of a memory address while at the same time

being able to specify an address to a program counter (PC) that will result in its containing an absolute address that the machine can recognize. For instance, if it was desired to change the contents of a 16 cell program counter from an absolute address of Word No. 0, say to Word No. 511, the following procedure could be used: The programmer would first specify an instruction that the CPU would decode as meaning "change the value in the program counter." (Such an instruction might be a JUMP instruction in a microcomputer's instruction set.) Following that instruction would be a word that held the desired value of the "low order address" or "word number within a page." Since a memory word only has eight cells, since eight cells can only represent 256 different patterns, and since one of the patterns is equivalent to a value of zero, then the largest number the eight cells can represent is 255. However, this is the largest number that is contained on a page. This value can be placed in the rightmost eight cells of a program counter. Now it is necessary to complete the address by getting the contents of another word from memory. Thus, immediately following the word that contained the "low address" would be another word that contained the "page number" of the address that the program counter was to contain. In this case the "page number" would be 1. When this value was placed in the remaining cells in the left portion of the program counter, the program counter would contain the following pattern:

```
*************************************************
*  *  *  *  *  *  *  *  |  *  *  *  *  *  *  *  *
*0*0*0*0*0*0*0*0*1|1*1*1*1*1*1*1*1*
*  *  *  *  *  *  *  *  |  *  *  *  *  *  *  *  *
*************************************************
```

FIGURE 5

If desired, the reader may verify, by using the formula presented previously for determining the decimal value of a binary number, that the pattern presented in Figure 5 corresponds to 511. Thus, by using a "page number" and a "word number" on a page (each of which will fit in an eight cell memory word), a method has been de-

monstrated that will result in the program counter being set to an absolute address for a word in memory. The following diagrams provide some examples as a summary.

```
PAGE NO. 0          WORD NO. 0
      ..                  ..
     .  .                .  .
......   ....... ......  .......
.                .  .            .
0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0
.                                .
................ ................
                .  .
                 ..
ABSOLUTE ADDRESS IN PC
```

1st physical word in Memory has an absolute address of: 0

```
PAGE NO. 0          WORD NO. 255
      ..                  ..
     .  .                .  .
......   ....... ......  .......
.                .  .            .
0 0 0 0 0 0 0 0  1 1 1 1 1 1 1 1
.                                .
................ ................
                .  .
                 ..
ABSOLUTE ADDRESS IN PC
```

256th physical word in Memory has an absolute address of: 255

```
PAGE NO. 1          WORD NO. 0
      ..                  ..
     .  .                .  .
......   ....... ......  .......
.                .  .            .
0 0 0 0 0 0 0 1  0 0 0 0 0 0 0 0
.                                .
................ ................
                .  .
                 ..
ABSOLUTE ADDRESS IN PC
```

257th physical word in Memory has an absolute address of: 256

```
PAGE NO. 1        WORD NO. 1
      ..                ..
     .  .              .  .
......   ....... ......  .......
.                .  .            .
0 0 0 0 0 0 0 1  0 0 0 0 0 0 0 1
.                                .
................ ................
                .  .
                 ..
ABSOLUTE ADDRESS IN PC
```

258th physical word in Memory has an absolute address of: 257

```
PAGE NO. 1        WORD NO. 2
      ..                ..
     .  .              .  .
......   ....... ......  .......
.                .  .            .
0 0 0 0 0 0 0 1  0 0 0 0 0 0 1 0
.                                .
................ ................
                .  .
                 ..
ABSOLUTE ADDRESS IN PC
```

259th physical word in Memory has an absolute address of: 258

```
PAGE NO. 1        WORD NO. 255
      ..                ..
     .  .              .  .
......   ....... ......  .......
.                .  .            .
0 0 0 0 0 0 0 1  1 1 1 1 1 1 1 1
.                                .
................ ................
                .  .
                 ..
ABSOLUTE ADDRESS IN PC
```

512th physical word in Memory has an absolute address of: 511

```
PAGE NO. 2        WORD NO. 0
      ..                ..
     .  .              .  .
......   ......  ......  ......
.                .  .          .
0 0 0 0 0 0 1 0  0 0 0 0 0 0 0 0
.                              .
................ ................
                .  .
                 ..
ABSOLUTE ADDRESS IN PC
```

513th physical word in Memory has an absolute address of: 512

```
PAGE NO. 3        WORD NO. 255
      ..                ..
     .  .              .  .
......   ......  ......  ......
.                .  .          .
0 0 0 0 0 0 1 1  1 1 1 1 1 1 1 1
.                              .
................ ................
                .  .
                 ..
ABSOLUTE ADDRESS IN PC
```

1024th physical word In memory has an absolute address of: 1023

FIGURE 6

By now the reader should have a pretty good understanding of the concepts regarding the organization of memory into electrical cells which can be in one of two possible states; the grouping of these cells into words which can hold patterns which the CPU can recognize as specifying particular operations; and the operation of a program counter which is able to hold the address of a word in memory from which the CPU is to obtain an instruction.

It is now time to discuss the operation of the scratch pad area of a computer - the accumulator (and some additional manipulating

registers found in most microprocessors).

As was pointed out earlier in this chapter, there is a section of a computer that is able to perform calculations and that can hold information while the CPU is in the process of fetching another instruction from the memory. The portion was termed an accumulator because it could accumulate information obtained as the CPU performed a series of instructions until such time as the CPU was directed to transfer the information elsewhere (or discarded it). An accumulator is also considered to be a mathematical REGISTER (series of cells) in a microcprocessor because it is the place where addition, subtraction, and various other mathematical operations (such as Boolean algebra) are actually performed under program control.

The concept of an accumulator is not difficult to understand, and its physical structure can be readily explained. The actual control of an accumulator by the CPU can be quite complex, but these complex electronic manipulations do not have to be understood by the computer user. It is only necessary to know the end results of the various operations that can be performed within an accumulator. The physical structure, as well as the kinds of operations that can be performed in the accumulator, will be discussed in this section.

An accumulator in a typical microprocessor can be considered as an eight-bit register similar to a word in memory except that the information in the cells can be manipulated in ways that may not be directly possible when the data is stored in memory.

```
B7  B6  B5  B4  B3  B2  B1  B0

*****************************
*  *   *   *   *   *   *   *
*1 * 0 * 1 * 0 * 1 * 0 * 1 * 0 *
*  *   *   *   *   *   *   *
*****************************
```

FIGURE 7

Figure 7 shows a collection of eight binary cells containing ones and zeros to represent an accumulator. The cells are numbered from left to right starting with B7 down to B0. The designations refer to bit positions within an accumulator. Note that the right-most cell is designated B0, and the eighth (left-most cell) is designated B7. The reader should become thoroughly familiar with the concept of assigning the reference of zero to the right-most bit position in a row of cells (similar to the concept of assigning a reference of zero to the first address of a word on a page in memory) as the convention is frequently used by computer technologists. The convention can be confusing for the beginner who fails to remember that the physical quantity is one more than the reference designation. The convention of labeling the first physical position as zero makes much more sense once the reader learns to think in terms of the binary numbering system. The reader must realize that the zero referred to so frequently in computer work when discussing actual operations actually represents a physical state (the status of an electronic switch) and does not necessarily imply the mathematical notion of "nothing." The concept of assigning a bit designation to the positions of cells within an accumulator will allow the reader to follow explanations of various accumulator operations.

One of the most fundamental and most often used operations of an accumulator is for it to simply hold a number while the CPU obtains a second operator. In most systems, the accumulator can be loaded with a value obtained from a location in memory or other special CPU registers. It then holds this value until it is time to perform some other operation with the accumulator. (It will become apparent later that an accumulator in most microprocessors can also receive information from external devices.)

Perhaps the second most often used function of an accumulator is to have it perform mathematical operations such as addition or subtraction. The operation is typically performed using the initial contents of an accumulator and the contents of a memory location or another CPU register. Thus, if an accumulator contained the binary equivalent of the decimal number 5, and an instruction to add it to the contents of a specific memory location that contained the binary equivalent of the decimal number 3 was encountered, the accumulator would end up with the value of 8 in binary form as shown in Figure 8.

```
 B7   B6   B5   B4   B3   B2   B1   B0

*********************************
*   *   *   *   *   *   *   *   *
* 0 * 0 * 0 * 0 * 0 * 1 * 0 * 1 *
*   *   *   *   *   *   *   *   *
*********************************
```

Original contents of the Accumulator

```
*********************************
*   *   *   *   *   *   *   *   *
* 0 * 0 * 0 * 0 * 0 * 0 * 1 * 1 *
*   *   *   *   *   *   *   *   *
*********************************
```

Contents of the specified word in memory

```
*********************************
*   *   *   *   *   *   *   *   *
* 0 * 0 * 0 * 0 * 1 * 0 * 0 * 0 *
*   *   *   *   *   *   *   *   *
*********************************
```

Final results after the addition in the accumulator

FIGURE 8

The next most frequently used group of operations for an accumulator is for it to perform Boolean mathematical operations between itself and/or other CPU registers or words in memory. These operations in typical units include the logical AND, OR and EXCLUSIVE OR operations. The operation of these Boolean mathematical functions will be described in more detail in another section of this book.

Another important capability of an accumulator is its ability to rotate its contents. In almost all systems, the contents of the accumulator can be rotated either to the right or left. This capability serves many useful purposes and is one method by which mathematical multiplication or division can be performed. Figure 9 illustrates the concept of rotating the contents of an accumulator.

```
 B7  B6  B5  B4  B3  B2  B1  B0

*****************************
*   *   *   *   *   *   *   *
* 0 * 0 * 0 * 0 * 0 * 0 * 1 * 0 *
*   *   *   *   *   *   *   *
*****************************
```

Original contents of the accumulator (equal to decimal 2)

```
*****************************
*   *   *   *   *   *   *   *
* 0 * 0 * 0 * 0 * 0 * 1 * 0 * 0 *
*   *   *   *   *   *   *   *
*****************************
```

Result when the accumulator is rotated to the left one time (value now equal 4)

```
*****************************
*   *   *   *   *   *   *   *
* 0 * 0 * 0 * 0 * 0 * 0 * 0 * 1 *
*   *   *   *   *   *   *   *
*****************************
```

Result when the accumulator is now rotated to the right two times (value now equal 1)

```
*****************************
*   *   *   *   *   *   *   *
* 1 * 0 * 0 * 0 * 0 * 0 * 0 * 0 *
*   *   *   *   *   *   *   *
*****************************
```

Note that if a rotate right command is done again that the 1 in position B0 will appear at B7!!

```
*****************************
*   *   *   *   *   *   *   *
* 0 * 0 * 0 * 0 * 0 * 0 * 0 * 1 *
*   *   *   *   *   *   *   *
*****************************
```

And that now a rotate left command would restore the 1 in position B7 back to B0!!

FIGURE 9

The astute reader may notice that the accumulator rotate capability also enables an accumulator to emulate a shift register which can be a valuable capability in many practical applications of the computer.

Finally, in most microcomputer systems, an accumulator may serve as a transfer register between the microcomputer and external devices. The reader may have noticed that up until now, no mention has been made of how information is transferred into or out of a microcomputer. Naturally, this capability is extremely important because a computer would be rather useless if people could not put information into the machine to have calculations or processing performed, and then receive information back from the machine to inform them of the results of the operations.

Communications between a microcomputer and external devices, whether those devices be simple switches, transducers, printing machines, cathode-ray-tube display units, keyboards, magnetic tape and disk systems, or whatever; are commonly referred to as input/output operations. They are collectively referred to in abbreviated form as I/O transfers.

While in most microcomputers the accumulator is the focal point of all I/O transfers, electronic buffers, typically referred to as I/O PORTS, are used to pass the information from an accumulator to an output device and vice-versa. The use of these I/O PORTS generally allows the microcomputer to be able to access a number of different devices by referring to specific input or output ports. Typical microcomputers have from a few to several hundred input and/or output ports. Some systems can reference an almost unlimited number of external devices by having devices connected to the system in such a manner that they appear to be a memory address. However, even in this situation, an I/O transfer operation typically will originate from, or terminate in, an accumulator register. It is important to note here that the typical I/O structure of a microcomputer allows a whole group of devices to be simultaneously hooked up to the microcomputer, and for the computer to receive information from, or transfer information to, a number of different devices as directed by a program. With appropriate programming, it is then possible to have the computer communicate with a large variety of devices in an essentially automatic mode. Thus, in one instance a microcomputer might be directed to receive information from an instrument such as a digital multimeter, and a fraction of a second later, output the result of the reading obtained, possibly after performing some arithmetic calculations, to a readout display. In a similar manner, a

microcomputer system can be configured to receive information as it is typed on a typewriter-like keyboard, and instantaneously have that information displayed on a cathode-ray-tube instrument.

An accumulator, however, is not the only register in a typical microcomputer in which data can be manipulated. Remember, though, that an accumulator is the most powerful type of register in a CPU. Most microprocessors have a number of additional types of registers within the CPU unit that can perform special types of operations.

One type of such a register is often referred to as a DATA register. Data registers in a CPU are generally provided so that temporary information, that is "data," may be held in a readily accessible position within the microcomputer system. One reason for doing this is that quite often a CPU is designed so that it can perform operations within itself considerably faster than performing the same operations with data that is in the memory banks of the computer or that might be available from an I/O device. In any event, these so-called data registers that are commonly provided in the CPU unit serve a number of useful purposes and in addition to simply holding data, they are also often capable of performing a limited number of operations on that data. For instance, these registers are often capable of incrementing, that is increasing the value of a piece of data being held in the register, or decrementing the value of a piece of data. Some microcomputers provide capability for data to be rotated or shifted within a data register. Occasionally, other types of operations might be possible. However, the most frequent use of data registers is for the simple purpose of holding data temporarily. Operations are then performed using the contents of those registers as well as those of the accumulator using the special general purpose capabilities of the latter register. It will be pointed out that generally the data registers within a CPU are eight-bit registers. However, in some machines, the data in two such registers can be manipulated simultaneously so that they can effectively process sixteen bits of data at a time. This is often a valuable feature to have available when a computer is performing mathematical operations involving medium size numbers, or when one is calculating memory addresses. Since most microprocessors are limited to a maximum of approximately 64,000 words of memory, virtually any memory address in such a system

can be expressed as a 16-bit binary number. That, of course, is precisely the number of bits available when two 8-bit registers are coupled together.

Another type of register that microcomputers possess might be termed a MEMORY POINTER register. Like a program counter register, a "memory pointer" register typically consists of two eight-bit registers coupled together in such a manner that they may represent a sixteen-bit address. This arrangement thus allows a memory pointer register to "point" to all the possible memory addresses within a microcomputer system. In some microcomputers, a memory pointer register is simply provided by giving special capabilities to a set of two data registers. Thus, in some instances the data registers may simply be used to hold information in the CPU and in other cases, the contents of two data registers taken as a whole are used to point to a location in memory. The distinction between the two types of uses is dependent upon the type of instruction that the microprocessor is directed to perform. It will also be mentioned here that some microcomputers have special kinds of memory pointer registers. These special types can have their contents augmented by values specified as part of certain classes of instructions. Such registers are generally referred to as INDEX registers.

One might be asking: "What is the difference between a program counter and a memory pointer or "index" register? After all, the main function of the program counter appears to be to "point" to memory. That is true, but remember, the function of the program counter is to inform the CPU where to obtain the next instruction to be executed. That is all. A memory pointer or index register, on the other hand, is used to tell the computer where it can obtain data or a piece of information stored at a particular address in memory. Note the crucial difference between the two types of registers. The program counter always tells the CPU where to obtain the next INSTRUCTION from memory, while a memory pointer register or an index register may be used to inform the CPU where to obtain a piece of DATA in memory. This arrangement thus allows a microcomputer to obtain information stored in a section of memory that is separate from the area in which the program directing the operation of the machine is stored.

Another type of register found in most microcomputers is referred to as a STACK POINTER. There are a variety of uses for a stack pointer register, some of which will be presented in later chapters. What does a stack pointer do? It simply points to a specific address in memory where information may be temporarily stored. The stack pointer register is usually designed so that each time the stack pointer register is utilized by some instruction, its value is either incremented or decremented according to the type of instruction being performed. This operation provides a means for the computer to PUSH information into a storage area in memory, or POP information from that storage area back into designated CPU registers. A frequent use of the stack pointer is to save the contents of the program counter when one branches from a main sequence of instructions to a minor sequence of instructions commonly referred to as a subroutine. When the instruction in the subroutine has been executed, the stack pointer register may be used to obtain the original value of the program counter from the storage area in memory so that the program counter is restored to its original value. The main portion of the program may then be continued from the point where the subroutine was originally referred to (called).

The type of registers being discussed at this point have been defined as typically consisting of a group of eight binary bits. The contents of these registers, which at any particular time contain a pattern of ones and zeros, represent DATA or MEMORY ADDRESSES. At various times when a microcomputer is in operation it is valuable to be able to ascertain certain types of information concerning the contents of various registers. What type of information may be of strong interest? Such things as whether or not the contents of a register are equal to zero, or less than or greater than that value. Or, whether the number of bits that are in the one state within a register are odd or even. Sometimes it may be valuable to know whether or not the result of a certain operation, such as adding the contents of two registers, results in an overflow condition, whereby the fixed length register is unable to represent the true results. In order to allow this type of information to be ascertained, and to give the microprocessor the capability to modify its operation as a function of the results it obtains when tests for such conditions are made on the contents of CPU registers (and/or the contents of locations in memory), microprocessors are equipped with a number of

indicators, commonly referred to as FLAGS. These FLAGS are nothing more than one-bit cells, or "flip-flops," that can assume a logic state of zero or one. Their state at any given time is determined by a set of rules defined by the microprocessor CPU designer to indicate the status or condition of selected CPU registers after certain types of operations. The condition of these special flip-flops may be "tested" by various types of instructions in order to determine their state. Then, based on their state, certain types of actions will either be taken, or not taken, thus giving the computer capability to make decisions as a program is executed. Since these flip-flops are actuated by certain conditions (as a function of operations that occur within selected registers) they can serve to "flag" an event. In other words, they may serve as a program attention-getting device.

The various types of FLAGS provided in a microprocessor are sometimes organized, or may at least be viewed as being organized, into a FLAG STATUS REGISTER. Each bit within the FLAG STATUS REGISTER may be used to represent a particular flag. Since most CPU registers contain eight bits, but there are rarely eight STATUS FLAGS provided in a microprocessor, the extra bit locations in a typical FLAG STATUS REGISTER may be simply unused, or used for special purposes, as designated by the CPU designer.

Some short descriptions and diagrams of the operation of FLAGS typically provided in a microprocessor are presented next. These brief examples serve to simply introduce the concepts at this time. There will be further discussion on the practical use of flags in other sections of this publication. Their introduction at this time serves simply to inform the reader of their presence, and to indicate their relationship to the contents of registers. The reader will later learn how these simple indicators greatly increase the capability of a computer.

There are four basic types of flags found on almost every micro-CPU manufactured. The first of these four commonly provided flags is called the ZERO FLAG. This flag is always set to a logic state of 1 if all of the bits in the register (being used to set the status of the flag) are set to 0. The ZERO FLAG is set to a 0 condition if any one of the bits in the register is a logic 1. Note that this relationship be-

tween the ZERO FLAG and the contents of the register is a "complemented" relationship. The ZERO FLAG is simply utilized to determine when the value of a particular register is zero. The diagram below illustrates the relationship between the ZERO FLAG and a register.

$\underbrace{0\,0\;0\,1\,0\;0\,0\,0}_{0}$	The ZERO FLAG is '0' if any bit in the register is a '1'.
$\underbrace{0\,0\;0\,0\,0\;0\,0\,0}_{1}$	The ZERO FLAG is '1' if the register value is '0'.

(The reader should realize that the status of the various flags in a CPU may be affected by a large number of different registers. That is, the execution of a variety of instructions involving any of a designated group of registers, may affect the status of the various flags. These relationships are all established by the designer of the CPU, and will be detailed in the manufacturer's literature.)

The second type of STATUS FLAG almost always provided in a microprocessor may be termed the SIGN FLAG. The SIGN FLAG is set to a logic state of 1 when the most significant bit (MSB) of the register is a 1 after certain types of instructions have been performed. The name of this flag derives from the practice of using a form of arithmetic known as "two's complement." This form of binary arithmetic uses the MSB in a register to designate the sign of the number represented in the remaining bit positions of the register. Conventionally, a 1 in the MSB designates the number as a negative number. If the MSB of the register is 0, the SIGN FLAG is set to 0, indicating that the number represented in the remaining bits of the register is positive in value. This convention results in the value of 0 being considered as a positive value. The diagram shown next provides a pictorial view of the relationship between the most significant bit in a register, and the status of the SIGN FLAG.

```
11 111 001
    1
MSB
00 000 111
    0
```

The SIGN FLAG is '1' if the most significant bit (MSB) in the register is '1'.

The SIGN FLAG is '0' if the MSB in the register is '0'.

The third type of status flag, found in essentially every CPU, is referred to as a CARRY FLAG. The CARRY FLAG may be considered as a one-bit (cell) extension of a register. The CARRY FLAG is generally associated with an accumulator register. This flag is set to a 1 position if the contents of the accumulator should overflow during an addition operation (or underflow during a subtraction operation). The CARRY FLAG is also sometimes utilized as an extension of the accumulator when the contents of the accumulator are shifted or rotated. When such is the case, the state of the CARRY FLAG will be altered as a function of the bit values it acquires during such operations. Some microprocessors also contain special instructions that may be used to set the CARRY FLAG to a predetermined state before performing operations that may affect its value. This allows tests to be conducted that will determine if the status of the CARRY FLAG has been altered by a series of operations. The diagram provided here illustrates the typical operation of the CARRY FLAG.

```
  10 101 010    ACC
  10 010 101    B
  -------------
1 00 111 111    ACC + B
↑
CARRY
```

If two numbers are added/subtracted, the CARRY FLAG will be set to '1' if an overflow/underflow condition occurs.

```
  10 101 010    ACC
  00 010 101    B
  -------------
0 10 111 111    ACC + B
```

If two numbers are added/subtracted, the CARRY FLAG will be set to '0' if an overflow/underflow condition does NOT occur.

The fourth flag that is generally found in all microprocessors is termed the PARITY FLAG. The diagram below illustrates that the PARITY FLAG is set to a 1 condition when the number of bits in the register that are in a logic 1 condition is an even value (without regard to the positions of the bits within the register). Thus, in an eight-bit register, the PARITY FLAG will be set to a 1 state if the number of bits set to the 1 condition is 2, 4, 6, 8, or 0 (0 usually being considered as an "even" value). The PARITY FLAG will be set to a zero if the number of bits in the register that are a logic 1 is an odd value (1, 3, 5, or 7) without regard to the position of the bits within the register. The PARITY FLAG is provided for the primary purpose of enabling the microprocessor system to be able to test for "transmission errors" as data formatted in a selected parity configuration is received by the CPU.

$$\underbrace{1\ 0\ 1\ 0\ 1\ 0\ 1\ 0}_{1}$$

The PARITY FLAG is typically set to a '1' condition if the number of bits set to '1' is even.

$$\underbrace{1\ 0\ 1\ 0\ 0\ 0\ 1\ 0}_{0}$$

The PARITY FLAG is thus set to the opposite state ('0' here) if the number of bits set to '1) is odd.

The four types of flags just described, as previously mentioned, may be found in just about every microprocessor available on the market. Some CPU devices may incorporate other types of STATUS FLAGS for other purposes. A type of status flag frequently encountered will be mentioned here. This flag is called the DECIMAL CARRY FLAG. This flag behaves in essentially the same manner as the standard CARRY FLAG previously discussed except that it typically is only actuated by a certain class of instructions referred to as decimal arithmetic commands. Furthermore, instead of operating to accept overflow or underflow from the most significant bit in an eight-bit register, it accepts the overflow or underflow from the fourth most significant bit in the register. The diagram below illustrates the DECIMAL CARRY FLAG being controlled by the associated bit within a register.

The DECIMAL CARRY FLAG is provided in some CPU devices. It is set or reset depending on whether a "carry" condition occurs out of the fourth bit in the accumulator register during certain types of operations.

At this point the reader has been introduced to the major fundamental parts of a microprocessor. The central processing unit - or "thinking" portion of a computer is able to manipulate data between various types of registers in the CPU such as the ACCUMULATOR (scratch-pad), MEMORY POINTER and DATA registers, and locations in the computer's memory banks. The manner in which data is manipulated is determined by having the CPU examine the binary codes contained in words in memory. Each binary code may represent a different type of operation that the computer is to perform. The binary codes to be interpreted are stored in sequential locations in the computer's memory banks. The CPU uses the "address" contained in the PROGRAM COUNTER register to keep track of where to obtain the next directive ("instruction") in memory. While each instruction directs the CPU to perform just one simple operation, by arranging sequences of instructions in memory, it is possible to "program" the computer to perform complex tasks. Additionally, since the execution of each directive takes but a tiny amount of time, a computer can perform thousands of individual steps in but a fraction of a second. This speed more than makes up for the step-by-step manner in which the computer must proceed and thus makes the machine a truly practical (indeed wonderful) device.

In the next chapter, a more formal presentation of the organization of a typical microprocessor CPU will be made. Then, the reader will be introduced to the major types of instructions that microprocessors are designed to be able to perform.

INSTRUCTIONS FOR MICROCOMPUTERS

This chapter will present the various types of instructions that a microprocessor is typically able to perform.

In the previous chapter, the reader learned the fundamental concept behind the operation of a computer. This was illustrated by using the analogy of a person obtaining directions written on pieces of paper stored in individual boxes. When necessary, calculations were performed using data obtained from specific boxes and the facilities of a scratch pad. The reader now knows that in a microprocessor system, the memory bank can store directions which are referred to as machine instructions, and the memory banks can also hold other types of information collectively referred to as data. The interpretation of instructions stored in memory is performed by the central processor unit, and the sequence of instructions to follow is controlled by a portion of the CPU called the program counter. The scratch pad portion of the CPU is the accumulator register. The accumulator register is often augmented by other auxiliary CPU registers, such as data registers, memory pointer registers, and index registers. These registers may be used to temporarily store information obtained from locations in memory or transferred to the CPU from devices external to the computer. Additionally, a memory pointer register and index registers are used to tell the CPU where to obtain specific pieces of data from storage in memory, or where to store information in memory. Remember, a memory pointer register points to information that is to be considered as data. The program counter register in the CPU always points to the next location in memory that is to be interpreted as an instruction by the CPU.

The only way a microprocessor can perform any useful function is for it to follow a series of instructions stored in the system's memory banks. And the only thing a properly operating CPU is able to do is to follow those explicit instructions. It follows, therefore, that if a person is going to have a microprocessor system perform useful functions, a portion of the microprocessor's memory banks must contain a series of carefully planned directives. These directives, as the reader now realizes, are encoded in the form of binary num-

bers, which, in this discussion, will be assumed to always consist of eight bits of information.

It is important to realize that while each instruction requires eight bits to be encoded, or, in other terms, is expressed as an eight-bit word, some types of instructions may need to be supplemented by one or more additional words immediately following an actual instruction word. The reason for this will come to light shortly.

When a microcomputer is initially placed into operation, it must be initialized to the proper location in memory from which to commence executing directives. This is accomplished by placing a memory address value into the program counter register of the CPU. How this is actually accomplished may vary from system to system, however, it is important to realize at this point that such a function must be performed whenever a computer is placed into operation.

Once the program counter has been set to an initial address, which indicates where the CPU may start to obtain instructions in memory, the computer's operation will be automatic as long as the sequence of instructions it follows is continuous. (Note that it is not necessary to have an infinite amount of memory in order to place a computer into a state of continuous operation! Such operation may be readily dictated in a very small amount of memory by simply placing directives in the instructional sequence that will cause the CPU to repeatedly perform the same group of instructions. Aspects of this type of operation will become apparent later.)

The reader will soon learn that the individual instructions that a microprocessor can perform seem quite insignificant when taken individually. Some readers may be amazed when they shortly discover that individual instructions that a microprocessor may execute are just about exactly of the same order of magnitude in capability as those given in the example using the post office boxes in the previous chapter. However, what the microprocessor lacks in individual instruction complexity is well made up for by the blinding speed in which it can perform each individual instruction. A typical microprocessor of today can execute between 50,000 to 1,000,000 or more individual instructions per second, depending on the particular CPU device being utilized! Thus, even though it may take

several thousand, or even tens of thousands, of instructions to perform a particular operation, the fact that all of these instructions can be executed in a mere split second gives the microcomputer its tremendous potential and problem solving capability. What is frightening at first glance though, is the realization that at some point in time (at least at this point in computer history), a human being must specify all of the individual instructions in the proper sequence, and place them in the proper locations in memory before a microprocessor system can be utilized to any practical advantage.

Readers who may have seen microprocessor systems merrily receiving information as it is typed in on a keyboard and displaying the results of calculations on a cathode-ray-tube display, or printing reams of text on a printer, may not have realized that behind that seemingly simple operation may have been hundreds and thousands of individual instructions. Each instruction is stored in a specific place in memory, in a precise sequence, and the slightest deviation from that prescribed sequence, or the alteration of even a single bit in any one of the instruction directives could cause total chaos to result. Other readers may have been introduced to this concept already, and have great fears that they themselves could not ever master such apparently complex and exacting procedures, and hence, be apprehensive that they could not possibly put a microprocessor to good use. Such is not the case. For one thing, not everyone that uses a microprocessor need necessarily know how to develop extensive machine language programs. (Machine language programs are those programs developed by the actual step-by-step process of specifying the machine codes and memory addresses using binary values (or similar numerical codes) in the same format as they are recognized and used by the CPU itself.) For one thing, it is now possible to obtain many types of programs that are already developed and ready for operation. One need simply load these programs into designated memory addresses within a microcomputer system, and commence to use the features or capabilities of the program at a level far removed from the details of machine language programming and requiring no knowledge of the individual machine coded instructions.

However, the purpose of this chapter is to give readers so interested a good idea of how a microprocessor is able to perform operations by following the sequence of instructions. In order to

make this presentation, it is necessary to first present the fundamental aspects of a microcomputer's operation, and then to discuss the basic types of directives or instructions that one may expect to find in a microcomputer capable of performing.

The fundamental aspect of a microcomputer's operation that needs to be explained at this point is as follows. The reader has learned that one of the key concepts behind a computer's operation is its ability to obtain a directive and then execute it. Note now, that this concept really consists of two parts. First, it must obtain the directive, and second, it must then execute that directive. The first part of this concept has been given quite a logical name by computer technologists. It is referred to as the FETCH cycle. The second part of the concept has just as logically been named the EXECUTE cycle. Therefore, one may consider the basic operation of a computer to take place in two parts - the FETCH part, and then the EXECUTE part. What happens during the FETCH part? The piece of information stored in the computer's memory banks at the address location specified by the contents of the program counter is fetched into the CPU. Once in the CPU, electronic logic determines what the binary code contained in the computer word means. Once this has been determined, the computer proceeds to execute the command it has just obtained. (The reader will learn further on in this section that sometimes a particular type of directive may call for additional fetch or supplementary operations. Regardless, the basic idea of each computer instruction being handled in two distinct operations consisting of a fetch and then execute cycle, is valid.)

When a microprocessor obtains a group of bits from a location in memory (word) during the fetch cycle, it will proceed to analyze the bit pattern contained in that word, and then perform the particular operation dictated by that bit pattern. Since an eight-bit word may represent up to 256 different patterns, one may see that it is possible for a microcomputer to be capable of performing, theoretically, 256 different types of directives. Actually, the number of different directives a microcomputer is capable of performing is a function of the type of architecture used by the designers. Typically, the number of instructions, or instruction types, that a microprocessor can execute is considerably less than the theoretical maximum of 256. On the other hand, a few microcomputers can execute considerably more

than 256 instructions. How is this possible? Remember the statement made earlier that sometimes an instruction may use additional words in memory. That is, a particular type of instruction may call for additional information to be contained in an adjacent memory word that will further classify the type of operation that is to be performed. At first glance, one may easily be struck with the thought that learning to develop programs for a microprocessor can lead to spending a lot of time and effort just learning about the various types of instructions that a CPU can execute. Indeed, that is the case if one wants to become a competent machine language programmer. On the other hand, it is possible to effectively utilize a microprocessor without becoming an expert machine language programmer. Indeed, this may well be the case for the majority of microprocessor users. Fortunately, while the actual numbers of unique instructions that a microprocessor may be capable of interpreting can be fairly large, in the hundreds of instructions, generally the multitude of possible instructions are simply permutations of a basic group. The purpose of this chapter will be to present and illustrate the basic types of instructions that one might expect a microcomputer to be able to perform. The reader will see that the various types of instructions whose concepts it would be important to learn are not all that staggering. As has been pointed out before, the whole key to understanding computer systems is to realize that the seemingly complex is actually based on repetition of relatively simple structures.

All of the many types of instructions that might be found in a microcomputer, which in cases can run into the hundreds, can actually be grouped into about half a dozen major categories. The permutations within each category can themselves reach one hundred or more individual instructions. The actual count for each type is quite often a function of the number of registers provided in the CPU unit. For example, one group of instructions might be called the LOAD, STORE, EXCHANGE group of instructions. These instructions all specify operations related to the transfer of information from locations in memory to specific CPU registers or vice-versa, or between CPU registers themselves. Naturally, if a particular CPU only has three or four different types of CPU registers, then the number of permutations of these instructions will be relatively small. On the other hand, a CPU device that has a dozen or so CPU registers can have a multitude of these types of instructions.

```
.............................
:   FLAGS    :    ACC      :
.............................
                :     B      :
                ............
                :     C      :
                ............
                :     D      :
                ............
                :     E      :
                ............
                :     H      :
                ............
                :     L      :
...................:.........
:    INDEX       : REGISTER  :
.............................
:    STACK       : POINTER   :
.............................
:   PROGRAM      : COUNTER   :
.............................
```

TYPICAL CPU REGISTERS

As an aid to the discussion that follows, a diagram illustrating the registers that might be found in a hypothetical, typical CPU is provided. The hypothetical CPU, for the purposes of this discussion, will have the following registers. An accumulator register, abbreviated by the letters ACC. Six DATA registers which will be referred to by the letters B, C, D, E, H, and L. Pairs of the DATA registers may be combined to form a memory pointer register. In the example illustration, registers B and C, D and E, and H and L, may be paired to form sixteen bit memory pointer registers. The diagram also shows that this hypothetical CPU contains a sixteen bit INDEX register which

can be viewed as consisting of two eight bit sections. Thus, the INDEX register can be initialized by loading the contents of eight bit registers into the left and right halves. The modal CPU also contains a sixteen bit STACK POINTER register organized in a similar manner, and, of course, a sixteen bit PROGRAM COUNTER. Also shown in the diagram is a register illustrated as an extension of the accumulator which has been labeled the FLAGS register. Information regarding this register will be presented later in this chapter. The reader should understand that different types of microprocessors may have more or less of these types of registers. For instance, some CPUs contain two accumulators. Other devices may have two, three, or more INDEX registers. Some CPU units do not have any DATA registers. The actual number and types of registers found within CPU devices is purely a matter of choice for the designers of that device. The discussion that follows, however, will provide most readers with sufficient background information to understand the capabilities of most microprocessors available.

To begin the discussion of the types of instructions that may be encountered in the microprocessor, the LOAD, STORE, and EXCHANGE types of instructions will be presented. These types of instructions are used to transfer information from locations in memory to CPU registers, or the reverse. They may also be used to transfer information from one CPU register to another CPU register. A LOAD instruction carries the connotation that information is to be transfered from memory to a CPU register, or from one CPU register to another CPU register. A STORE instruction may be used to denote the operation of transferring information from a CPU register to a location in memory. While one could view a similar type of operation or transfer of information from one CPU register to another, in order to avoid confusing the reader, the STORE type of instruction will always be referred to in this manual as indicating a memory reference command. That is, the data is to be transferred from a CPU register to a location in memory. The case of data being transferred amongst CPU registers will always be referred to as a load directive (as will the transfer of data from memory to CPU registers). This is because the concept of storing the contents of one CPU register in another CPU register is the same as loading data obtained from one CPU register into another CPU register. It is important, however, to reserve the distinction between loading information from memory

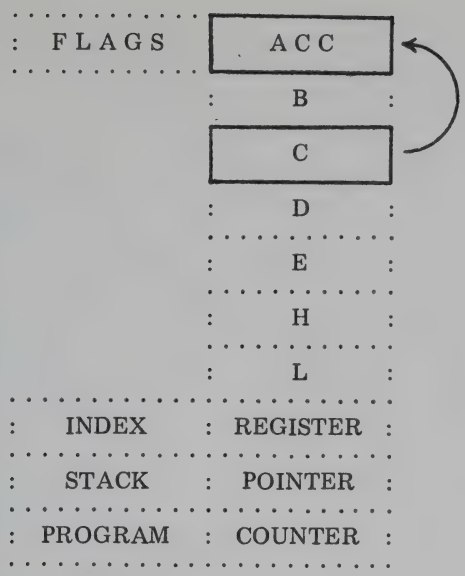
to the CPU device or storing information from the CPU device into locations in memory, hence, the naming of the two basic types of transfer operations.

Another type of operation that may be defined in this group of instructions can be called the EXCHANGE type. The EXCHANGE type of instruction is quite often provided as part of a microprocessor's instruction set for transferring information between CPU registers. It may actually be provided for transferring information to and from locations in memory and CPU registers in relatively sophisticated microprocessors. Note, however, in cases where it is not, one may achieve essentially the same type of operation by the proper combination of load and store instructions. An EXCHANGE instruction, for instance, may be used to swap the contents of two CPU registers. Some microprocessors do not provide for this type of instruction.

It may be interesting for the reader to note that if an actual EXCHANGE type of instruction is not provided, one could achieve the equivalent command by combining several load type operations. However, it requires the use of an extra CPU register. For example, an EXCHANGE instruction would allow the contents of two CPU registers, such as B and C to be swapped. If an exchange type of command was not available, an actual exchange type of operation could be accomplished if the contents of register B were first transferred to the accumulator. Then the contents of register C could be transferred to register B. Finally, the contents of the accumulator could be transferred to register C. These transfers could be accomplished using the LOAD instructions.

An example of the action of a LOAD instruction taking place within the CPU from register C to the accumulator is illustrated next.

When a LOAD instruction is used to transfer information from a location in memory to a CPU register, one must take into consideration another aspect of a microcomputer's operation. That is, determining the location or address in memory from which the data is to be fetched. One way in which the location in memory may be specified is through the use of a memory pointer register. The next

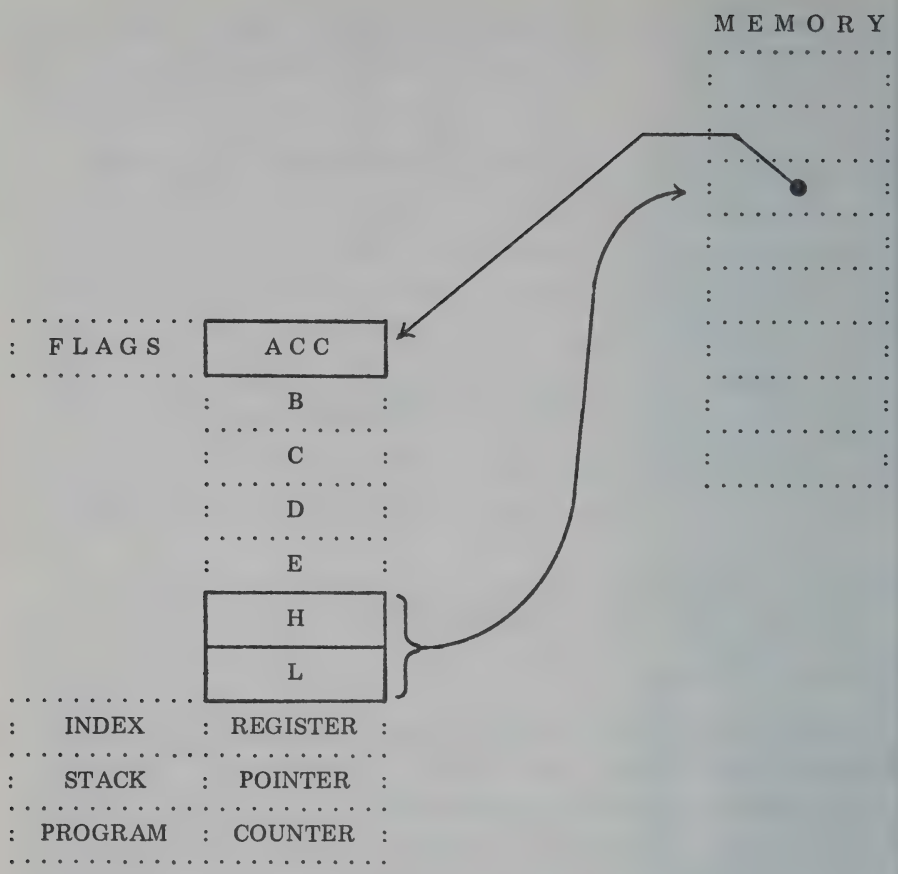


LOAD The Accumulator
From Register "C"

diagram illustrates how (through previous instructions) the H and L CPU registers, operating as a memory pointer register, could be set up to specify a particular memory address. A LOAD instruction could then specify that the data in the memory location specified by the present contents of the H and L registers was to be loaded into a particular CPU register. In the example, the data is being transferred into the accumulator register.

Similarly, another diagram illustrates how CPU registers D and E might be used as a MEMORY POINTER register to specify the location in memory where data was to be deposited in memory when a STORE instruction was executed. In this example, the information being stored was obtained from the accumulator register.

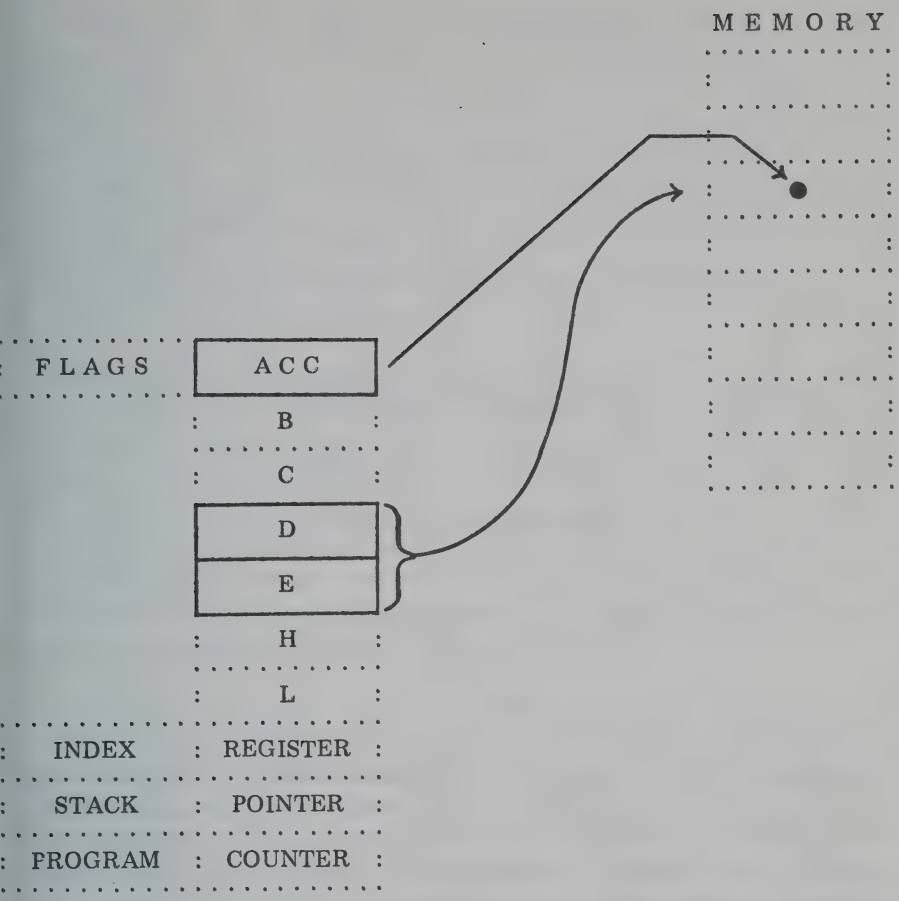
EXCHANGE type instructions may provide for single or double register transfers. The diagram provided next illustrates a hypothetical double register exchange instruction. In this example, the contents of registers D and E are being exchanged with the contents of



LOAD the Accumulator from Memory

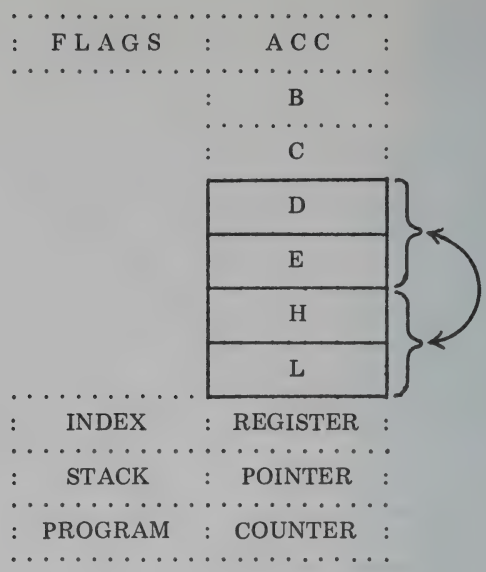
H and L. This would be the case when a CPU device had provisions for exchanging the contents of MEMORY POINTER registers using one directive.

It may now be apparent that the number of individual LOAD, STORE, and EXCHANGE instructions for a typical microprocessor is highly dependent on the number of CPU registers available in the machine. Remember, each instruction indicating the use of different



Store the Accumulator in Memory

registers will require a different, unique machine code which is the binary representation that the computer can recognize as an instruction. Thus, an instruction to "load the accumulator with the contents of CPU register B" will have a different code than that for "load the accumulator with the contents of CPU register C." Similarly, the instruction to "load CPU register B with the contents of the accumulator," and the "loading of CPU register C from the accumulator" will also be unique. The hypothetical CPU device being used for illustra-



Exchange the Contents of H & L
with the Contents of D & E

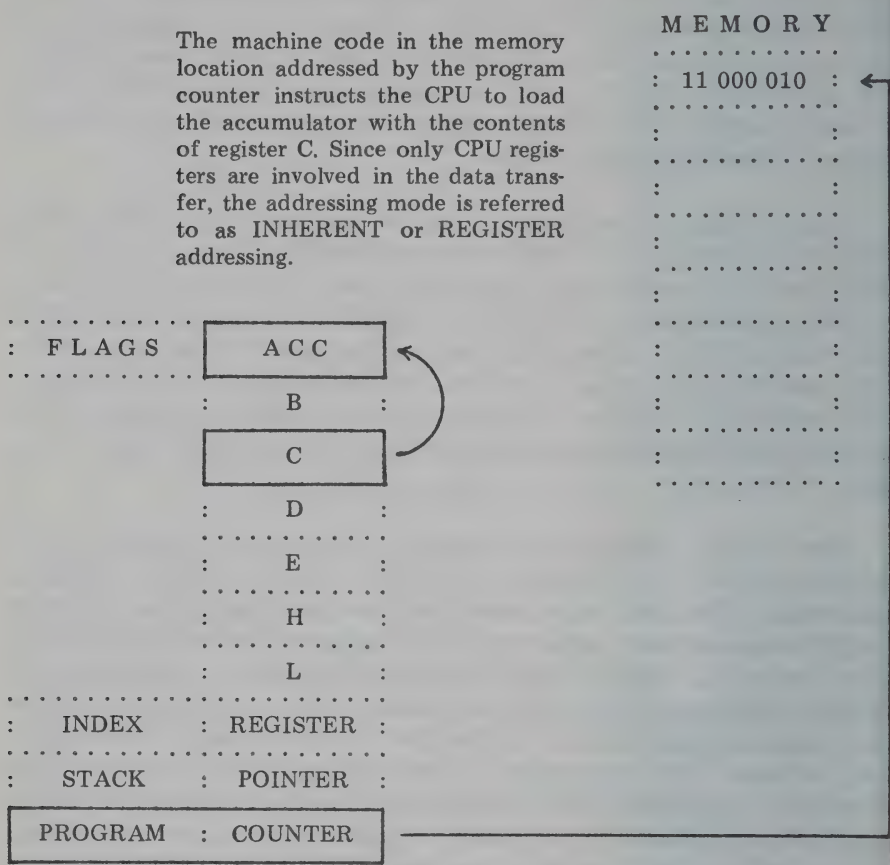
tive purposes would require 36 different commands just to represent all the possible single register LOAD transfers amongst the available CPU registers. A CPU that had ten different DATA registers would require 100 different instructions if all possible inter-CPU register load operations were to be permitted. Naturally, adding in the directives to permit the transfer of information with locations in memory, and single or double register exchange instructions increases the number of unique codes that must be represented by the instruction set. The complexity of a microprocessor's instruction set, the reader may now be beginning to recognize, will be highly dependent on just how complex the particular CPU device is. However, the addition of extra CPU registers, while giving more power to a device and requiring a larger instruction set, can still be managed quite comfortably when one realizes that the basic types of instructions still remain the same. One may just as easily learn to LOAD, STORE, and EXCHANGE the contents of CPU registers when there are two CPU DATA registers or ten such registers.

Another factor that enters into determining the number of unique machine codes that may be required to completely specify the instruction set of a microprocessor is connected with what may be referred to as addressing modes. When, for example, an instruction specifies a load operation, it is necessary for the computer to know the location from which data will be obtained, or the originating register, and the location to which the data will be transferred, which is the destination register. Most microprocessors have a variety of addressing modes and different types of instructions will utilize one or more of these addressing modes.

Some addressing modes are so simple that one may hardly realize that an addressing mode has been specified. For instance, a load instruction that merely moves data from one CPU register to another utilizes an addressing mode that is commonly termed INHERENT addressing. This is because the instruction itself inherently specifies any addressing information required by the instruction. For instance, the instruction load the accumulator from register C contains all the addressing information required for the CPU to complete the execution of the directive.

The following diagram illustrates the hypothetical operation of a microcomputer as it processes an instruction that utilizes inherent addressing. The instruction being processed is: Load the accumulator with the contents of register C. The diagram helps the reader to see that addressing modes being discussed here refer to references to DATA that the computer is manipulating. Thus, the diagram shows that the program counter in the CPU holds the value of the memory address which contains the instruction code that the CPU is currently processing. The instruction currently being processed, might be represented in binary machine code by the pattern 1 1 0 0 0 0 1 0. The actual machine code that the CPU interprets to execute an instruction is often organized into formats that a computer programmer can quickly learn. (After all, microprocessors are designed by human engineers who must devise some logical pattern by which they can construct the machine.) In the diagram illustrated, one could hypothesize that the first two binary bits of information (reading from left to right) in an instruction word, when they were both in the '1' condition, indicated that a load operation was to be performed. The next three bits could be used to represent the

destination register. For instance, if all three destination bits were in the zero state, it could signify that the destination register was the accumulator. Finally, the last three binary digits making up an in-



Inherent or Register Addressing

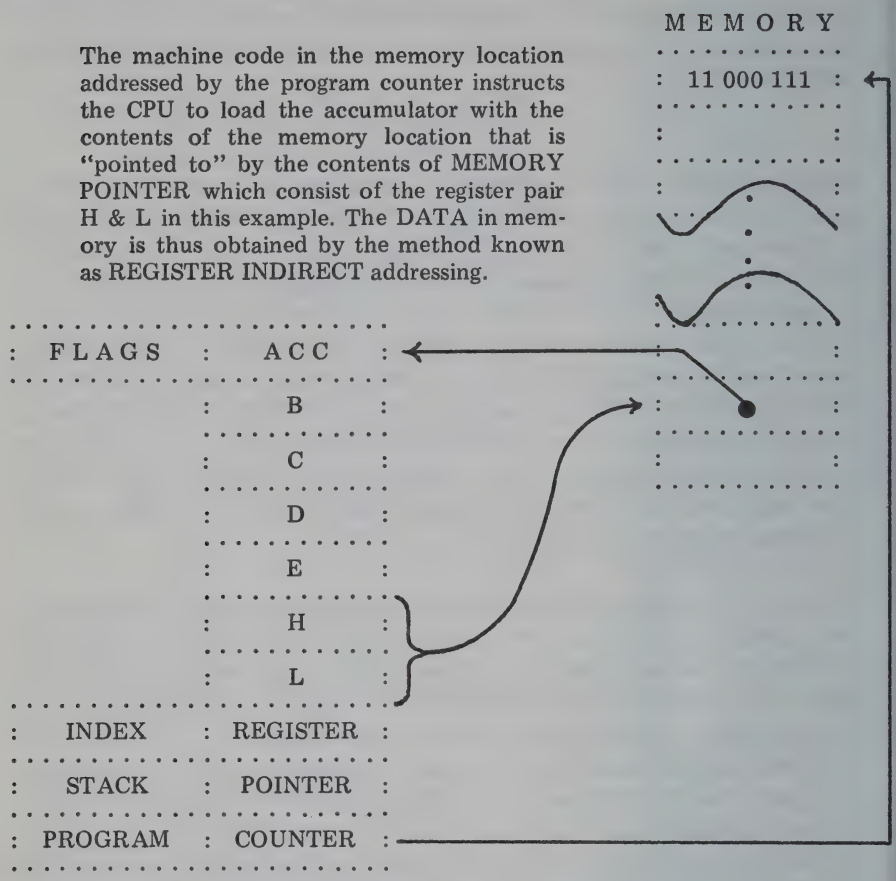
struction byte could be used to indicate the originating register. In the example, the pattern 0 1 0 specifies that CPU register C is the originating register for the data transfer. The diagram thus shows that

when the computer encounters that pattern, 1 1 0 0 0 0 1 0 as an instruction, it will recognize it as a directive to load the contents of register C into the accumulator. The addressing information for the movement of the data is contained entirely within the bit pattern which picks up the actual instruction; thus the addressing mode is INHERENT.

What happens in the case when one wants to load data from a location in memory to a CPU register? A few pages earlier, a diagram was presented that showed data being loaded from a location in memory to the accumulator. From what location in memory was the data obtained? The diagram indicated that the address in memory from which the data was obtained was pointed to by the contents of the H and L CPU registers. What type of addressing mode was being utilized in that example? The addressing mode utilized is commonly referred to as REGISTER INDIRECT. The following diagram illustrates that type of operation being performed in a little greater detail so that the reader may clearly discern the relevance of the addressing mode being utilized.

As in the previous illustration, the diagram shows the program counter pointing to an address in memory that contains the machine code for the actual load instruction being executed. In a manner similar to the diagram just presented, it may be hypothesized once again that the first two binary bits in the instruction word, when in a '1' condition, indicate that a load instruction is to be executed. Once again, the next three bits in the instruction word, which represent the destination register, are in the '0' condition which, for this example, specifies the accumulator register. The last three bits of the instruction word represent the source register. One could assume that when all three of the remaining bits were in a '1' condition, the source register was to be a location in memory. Fine. What location? A typical microprocessor might have thousands of words of memory in the system. Well, the CPU designers could, for example, have constructed the CPU so that whenever it encountered that particular instruction code: 1 1 0 0 0 1 1 1, it would "know" that it was to obtain the data from the address in memory pointed to by the contents of the H and L MEMORY POINTER registers. Obtaining the data from an address in memory utilizing the technique of accessing

an address via memory pointer registers is called REGISTER INDIRECT addressing.

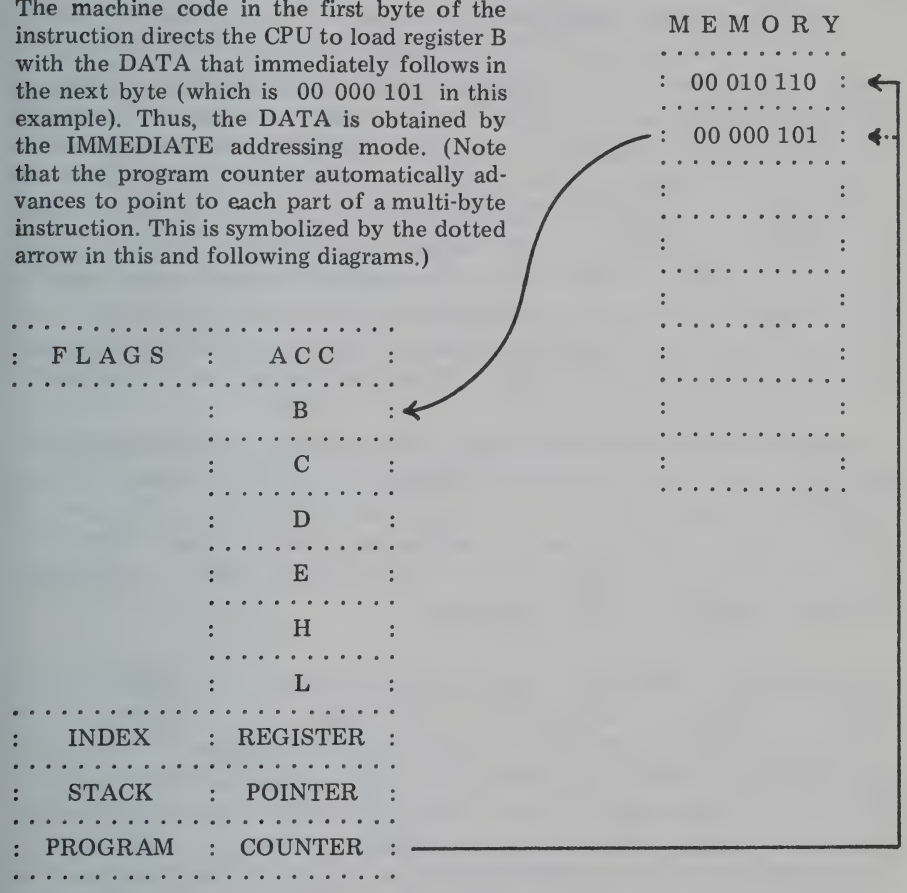


Register Indirect Addressing

IMMEDIATE addressing is another mode by which data in a memory location may be manipulated. IMMEDIATE addressing also introduces a new concept that has not been discussed to this point. That is the fact that a computer instruction may utilize more than

one word in memory. An instruction that specifies IMMEDIATE addressing generally requires two consecutive words in memory. The first word may be considered the actual instruction. Upon encountering an instruction that the CPU decodes and determines to be in the IMMEDIATE addressing classification, the CPU will automatically realize that in order to complete execution of the instruction, it will have to obtain a DATA byte from the next location in memory. (The program counter will advance and the computer will

The machine code in the first byte of the instruction directs the CPU to load register B with the DATA that immediately follows in the next byte (which is 00 000 101 in this example). Thus, the DATA is obtained by the IMMEDIATE addressing mode. (Note that the program counter automatically advances to point to each part of a multi-byte instruction. This is symbolized by the dotted arrow in this and following diagrams.)



Immediate Addressing

actually perform another FETCH and EXECUTE cycle to obtain the next consecutive word in memory.) However, the CPU will remember that the information it obtained from the second byte is actually to be used as DATA. The practice of following certain types of instructions immediately by a piece of DATA that the computer is to utilize as an operand for the instruction is termed IMMEDIATE addressing.

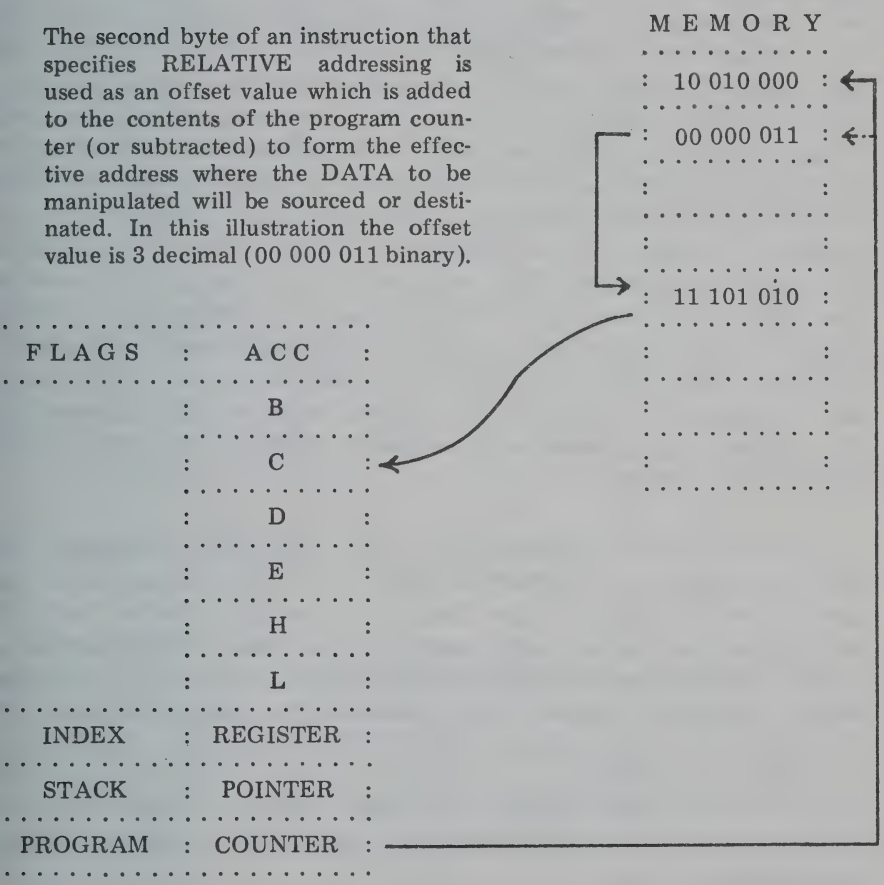
The diagram illustrating IMMEDIATE addressing highlights the important aspects of the operation. In the illustration, the first byte of the two-byte instruction contains the binary pattern 0 0 0 1 0 1 1 0. A model CPU could be assumed to interpret the bit pattern illustrated in the diagram as an IMMEDIATE addressing mode command that was to load the data found in the next memory location into CPU register B. The diagram helps to illustrate how the DATA WORD containing the binary value 0 0 0 0 0 1 0 1 would be transferred into register B. The diagram also points out that in order to completely execute the instruction, the program counter would have to successively advance from the original instruction byte to the following data byte (which would automatically occur as soon as the CPU analyzed the binary bit pattern and the electronic logic determined that an IMMEDIATE type of instruction was being executed).

There are more ways by which a microprocessor can transfer information to and from locations in memory and CPU registers. The discussion of how this may be accomplished through the use of different types of addressing modes will be continued with the presentation of a slightly more complex addressing mechanism referred to as RELATIVE addressing.

RELATIVE addressing may be considered as a slightly more advanced version of IMMEDIATE addressing. Now, instead of the second byte of an instruction containing the data that is to be manipulated by the instruction, that word will contain a value that is to be added or subtracted to the current value of the program counter. That new value, which may be referred to as the EFFECTIVE ADDRESS (EA), is the address from which the actual data is to be obtained. Note that this is really quite similar to an IMMEDIATE type of instruction, except that now the range of addresses in which the data byte may reside has been extended from simply being one

ocation (the location immediately following the instruction byte) o a range of addresses centered about the instruction.

The diagram illustrating RELATIVE addressing will clarify the oncept. As in the previous diagram referring to IMMEDIATE ddressing, one may observe that the entire instruction requires two ytes in memory, and will actually be performed by the CPU going hrough two complete fetch and execute cycles as the program ounter is advanced for each byte. Now, however, instead of the



Program Counter Relative Addressing

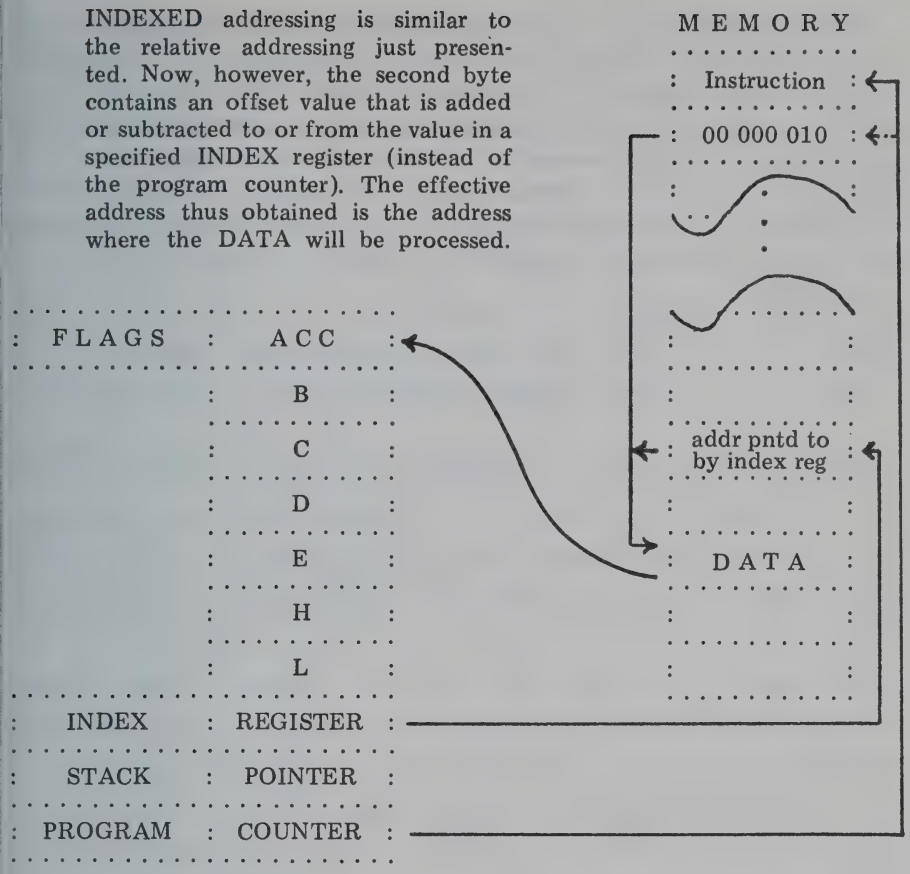
second byte of the instruction containing the DATA byte, it simply contains a value which is to be added (or subtracted) to the contents of the program counter. This new value forms an EFFECTIVE ADDRESS (EA) which indicates the memory address from which the actual DATA BYTE is to be obtained. The example then shows the data byte (consisting of the pattern 1 1 1 0 1 0 1 0) finally being transferred into CPU register C.

At this point, the reader might not readily see any advantage in having a RELATIVE addressing mode over the IMMEDIATE addressing mode. At fact, it might seem that the RELATIVE mode was considerably less efficient than the IMMEDIATE mode because one actually used three bytes of memory in order to complete the transfer process. Indeed, some microprocessors do not have a RELATIVE addressing mode. Suffice it to point out at this time, though, that the addressing mode is valuable when it can be incorporated. An inkling of its value may be gleaned by noting that the RELATIVE addressing mode permits the data byte stored at one location in memory to be referred to by a number of instructions contained in a program that may be in the vicinity of the address that contains the data byte. The significance of having this capability cannot be appreciated until one has actually learned to program a microprocessor. Take it for granted at this point that the addressing mode is sufficiently valuable to cause the designers of microprocessor CPU chips to spend considerable effort to include such capabilities in many of the devices.

Another type of addressing is called INDEXED addressing. INDEXED addressing is similar to RELATIVE addressing except that now the second byte of the instruction contains a value that is to be added or subtracted to the current value of the INDEX register in order to form the EFFECTIVE ADDRESS where the DATA is residing. The only real difference between the RELATIVE and INDEXED addressing modes is that the INDEX register replaces the PROGRAM COUNTER as the base address. This base address will then be combined with the offset address contained in the second byte of the instruction to form the actual address of where the data is residing.

The following diagram illustrates the concept of INDEXED

INDEXED addressing is similar to the relative addressing just presented. Now, however, the second byte contains an offset value that is added or subtracted to or from the value in a specified INDEX register (instead of the program counter). The effective address thus obtained is the address where the DATA will be processed.



Indexed Addressing

addressing. The reader may compare this diagram with the previous diagram to discern the fundamental difference between the types of addressing being discussed. In the example illustration one may observe that the value 0 0 0 0 0 0 1 0 binary (two decimal) contained in the second byte of the instruction is added to the value in the INDEX register to form the final address value, which is effectively the contents of the INDEX register plus two. The content of the memory word at the EA (effective address) is then transferred, in this example, to the accumulator (as would be the case if a LOAD instruction was being executed).

The reader may note that the INDEXED addressing mode only exists when the microprocessor CPU is equipped with an INDEX register. Some commonly available microprocessors do not have INDEX registers! INDEX register equipped microprocessors are particularly valuable when one needs to move relatively large amounts of information to and from various sections in memory. Such as when one desires to use a microcomputer as a word processing or text editing tool. Or, when one intends to do a goodly amount of search or sorting operations. CPU devices that do not have so-called INDEX registers can generally perform the same functions (although in a less efficient manner) using an alternate addressing mode, such as REGISTER INDIRECT addressing which has already been described.

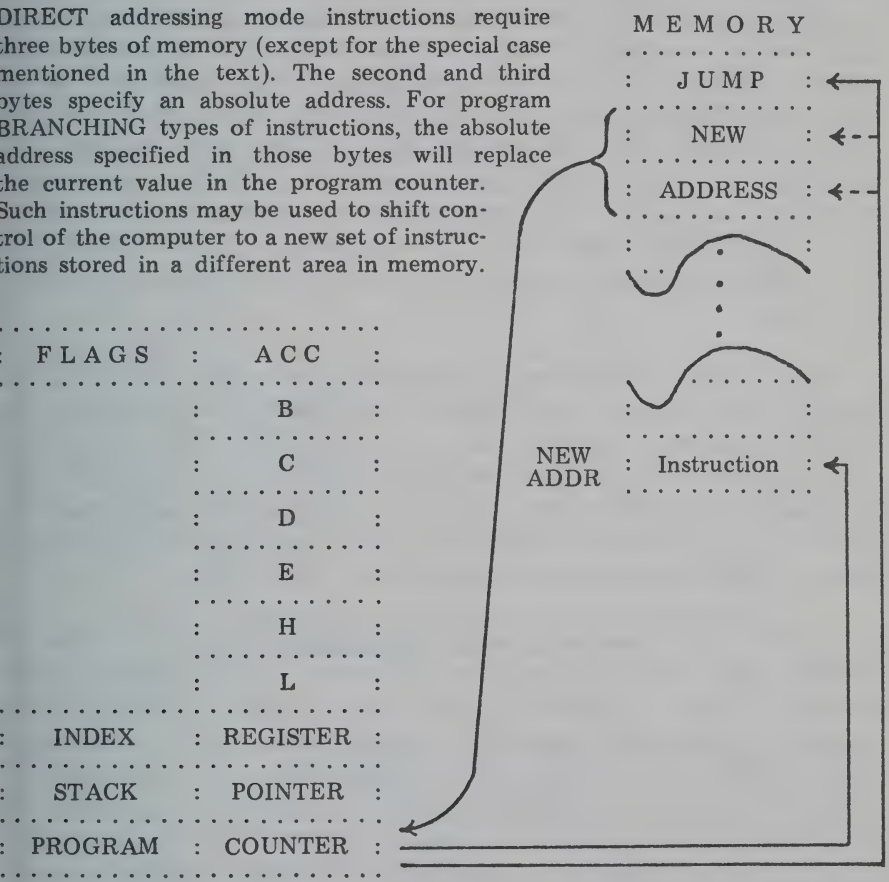
Some microprocessors that do have one or more INDEX registers, and can thus utilize INDEX addressing techniques, may have an additional addressing mode, referred to as AUTO-INDEXED addressing. This addressing mode operates exactly like regular INDEX addressing except that whenever an instruction is executed using the AUTO-INDEX mode, the INDEX register referred to by the instruction will have its value incremented (or decremented) automatically. In some machines, the value by which the contents of the INDEX register is incremented (or decremented) is fixed. In such cases, the increment value is usually one. However, in other machines the increment value may be specified in the second byte of the instruction in place of the index offset value. In such cases, the value of the INDEX register alone will serve as the effective address for the instruction, and the value contained in the second byte will indicate the value by which the contents of the INDEX register is to be altered automatically.

Machines equipped with AUTO-INDEX capability provide additional effeciency when executing programs that involve the transfer of relatively large amounts of data. However, the same type of operation can be effectively emulated on machines that do not have the automatic indexing capability by simply incrementing or decrementing the value contained in the INDEX register using appropriate instructions of a class that will be described further on in this chapter.

The next mode of addressing to be discussed is found on most

microprocessors. This form of addressing is known as DIRECT addressing. When this mode of addressing is used, the actual address is included as part of the instruction. That is, the instruction will be multi-byte, and those bytes after the actual instruction code are used to specify the absolute memory address associated with the instruction. In most eight-bit machines, the address portion of the instruction will require two bytes. This allows up to 64 K words of memory to be directly referenced. DIRECT addressing is almost

DIRECT addressing mode instructions require three bytes of memory (except for the special case mentioned in the text). The second and third bytes specify an absolute address. For program BRANCHING types of instructions, the absolute address specified in those bytes will replace the current value in the program counter. Such instructions may be used to shift control of the computer to a new set of instructions stored in a different area in memory.



Direct Addressing Illustrated for a Program Counter Modifying Instruction such as a JUMP or CALL Directive

always used for a special class of instructions referred to as JUMP and CALL directives. This is because these types of instructions require altering the contents of the CPU's PROGRAM COUNTER register. In other words, to execute the instruction, a new address contained in the address bytes of the instruction will be loaded into the program counter, thus causing program execution to continue in a new section of memory.

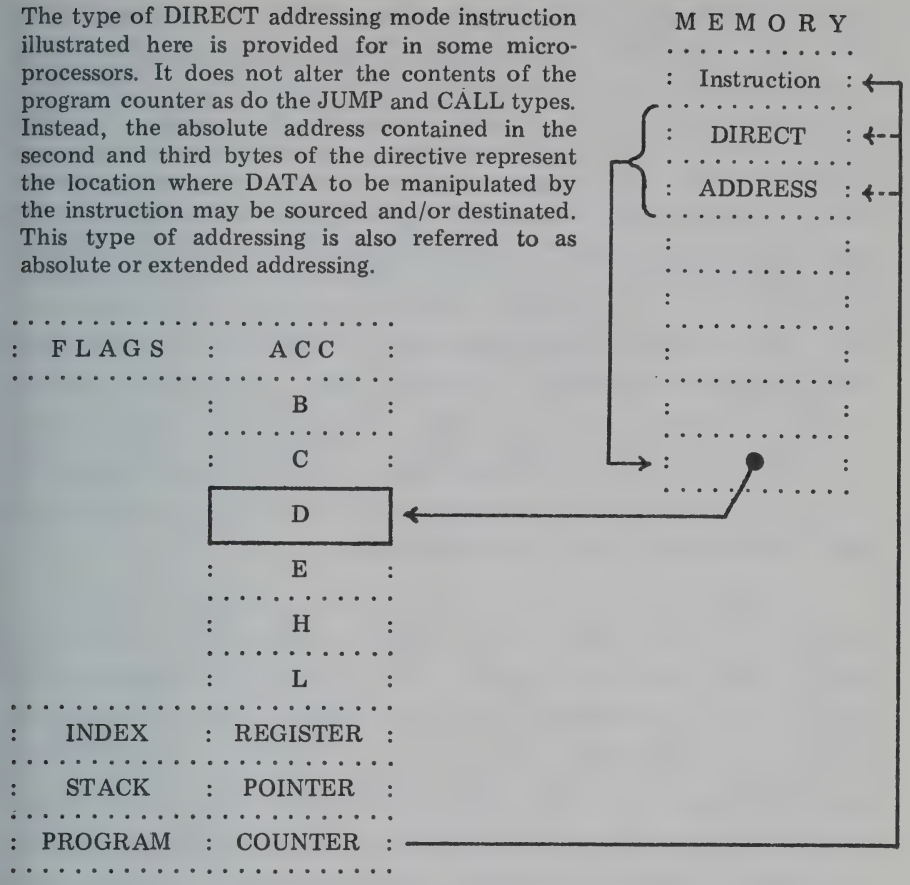
Of course, some kinds of microprocessors allow other types of instructions (other than the JUMP and CALL instructions mentioned) to utilize the DIRECT addressing mode. In such cases, the address contained in the address bytes of the instruction is the absolute address where DATA being manipulated by the instruction is located. The diagram below, as contrasted with the diagram just presented, presents a pictorial representation of this type of operation. Note that in this illustration the PROGRAM COUNTER operates in its normal manner. In the illustration above, representing the execution of a JUMP instruction, the contents of the PROGRAM COUNTER are altered at the conclusion of the execution of the instruction.

While in most machines, instructions that utilize DIRECT addressing will require three bytes, in a few machines two byte DIRECT addressing instructions may be possible. When this is the case, the second byte of the instruction is used to specify an absolute address that is within the first 256 (decimal) bytes of computer memory. This is often provided for in microprocessors that utilize a memory addressing structure referred to as PAGING.

PAGING refers to the practice of dividing the allocated range of memory addresses into blocks of addresses. The number of words within a block is defined by the system designer. Normally, for technical reasons, the number of words chosen to be counted as a block is 256. However, in special designs this number may be different. For virtually all practical systems, however, the number of words specified in a block will be some power of two, such as 256, 512, or 1024. These blocks of words are figuratively referred to as PAGES of memory.

The reader will probably see and hear many people refer to

The type of DIRECT addressing mode instruction illustrated here is provided for in some microprocessors. It does not alter the contents of the program counter as do the JUMP and CALL types. Instead, the absolute address contained in the second and third bytes of the directive represent the location where DATA to be manipulated by the instruction may be sourced and/or destinated. This type of addressing is also referred to as absolute or extended addressing.



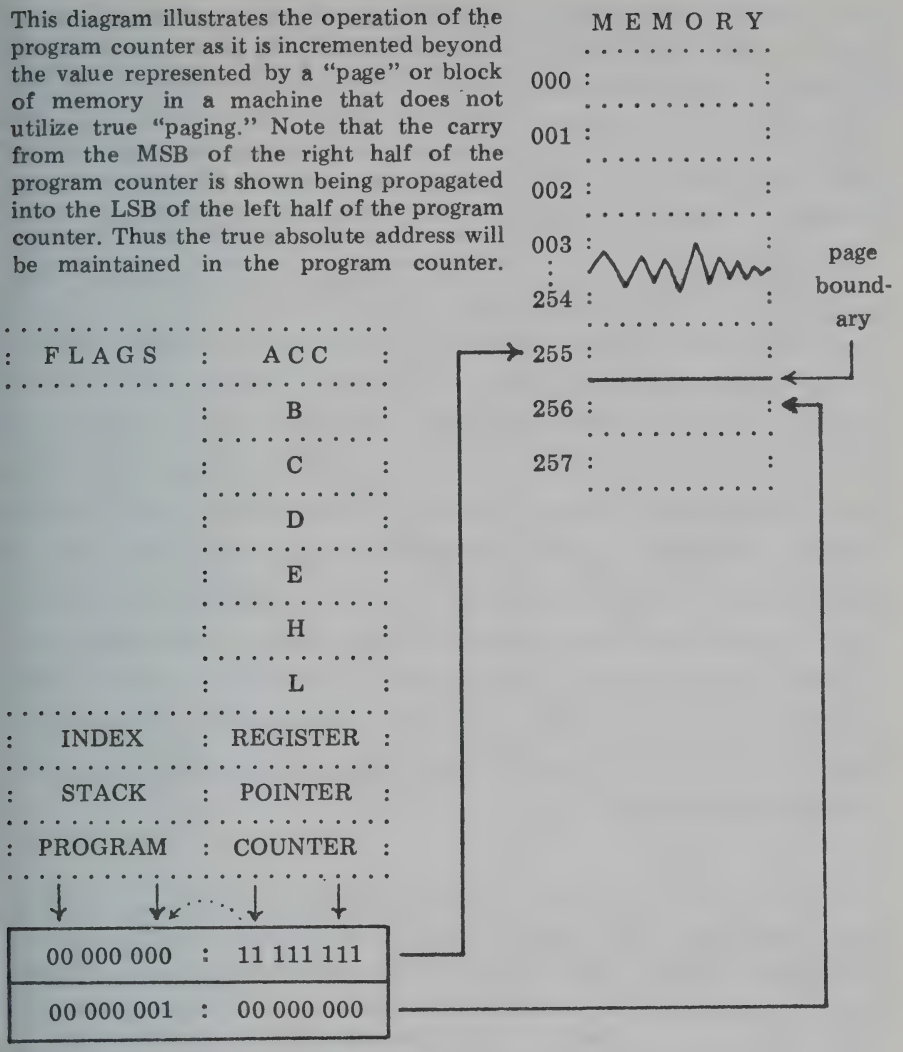
Direct Addressing Illustrated for a Data Manipulating Type of Instruction

PAGING in a microprocessor in the context of referring to any defined block of memory addresses. For example, one might find that a particular manufacturer's reference book would specify that relative addressing is limited to one page of memory, meaning that the second byte of an instruction utilizing the RELATIVE addressing mode may be coupled with the value of the PROGRAM COUNTER to form an effective address within a block of, for instance, 256 words of memory centered about the current value of the program

counter. This usage, in the opinion of some, may be misleading if the actual processor is not architecturally designed as a PAGED machine. A true PAGED machine operates in such a manner that it requires a special directive in order for the PROGRAM COUNTER to proceed across a PAGE boundary. For example, if a particular CPU is a true PAGED machine, that allocated 256 words of memory per PAGE, and the machine was capable of addressing 64 K words of memory, the program counter would be made up of two eight-bit registers side-by-side. One of the eight-bit registers would be used to indicate the PAGE or block in memory that was currently being utilized by the computer. The second register would then indicate the location on a PAGE (within a block). In a true "paged" machine, the program counter will not advance the "page" portion of the program counter when the "location on a page register" cycles around past its maximum value. Stated another way, there is no carry from the most significant position of the "location on a page register" to the least significant position of the "page" register.

The distinction between a processor that utilizes true paged addressing and one that does not may be more clearly discerned by studying the next two diagrams. The first diagram illustrates the operation of an addressing register, such as the program counter register, in a machine that does not have true paging. (However, since the processor does consider the program counter to consist of two eight-bit sections, one could arbitrarily refer to the eight-bit portion containing the most significant bits as the PAGE register, indicating the block of memory currently being processed. The right hand group of bits will then represent the particular location within a block, or PAGE. This relationship is often referred to using the nomenclature PAGING even though the machine does not operate in the true paging fashion which will be presented shortly.) The diagram for this example has been labeled PSEUDO-PAGING, which is an apt definition. The diagram illustrates the action of the program counter in a machine that does not utilize true paging as the value in the program counter is incremented beyond the value defined as a block of memory addresses. When the least significant eight-bits in the sixteen-bit wide program counter (which may be considered as two eight bit length registers) contains the binary value 1 1 1 1 1 1 1 1, and the value is incremented by the count of one, the new value becomes 1 0 0 0 0 0 0 0 0 binary. In the diagram labeled Pseudo-

This diagram illustrates the operation of the program counter as it is incremented beyond the value represented by a "page" or block of memory in a machine that does not utilize true "paging." Note that the carry from the MSB of the right half of the program counter is shown being propagated into the LSB of the left half of the program counter. Thus the true absolute address will be maintained in the program counter.



Program Counter Operation in a Pseudo-paging Machine

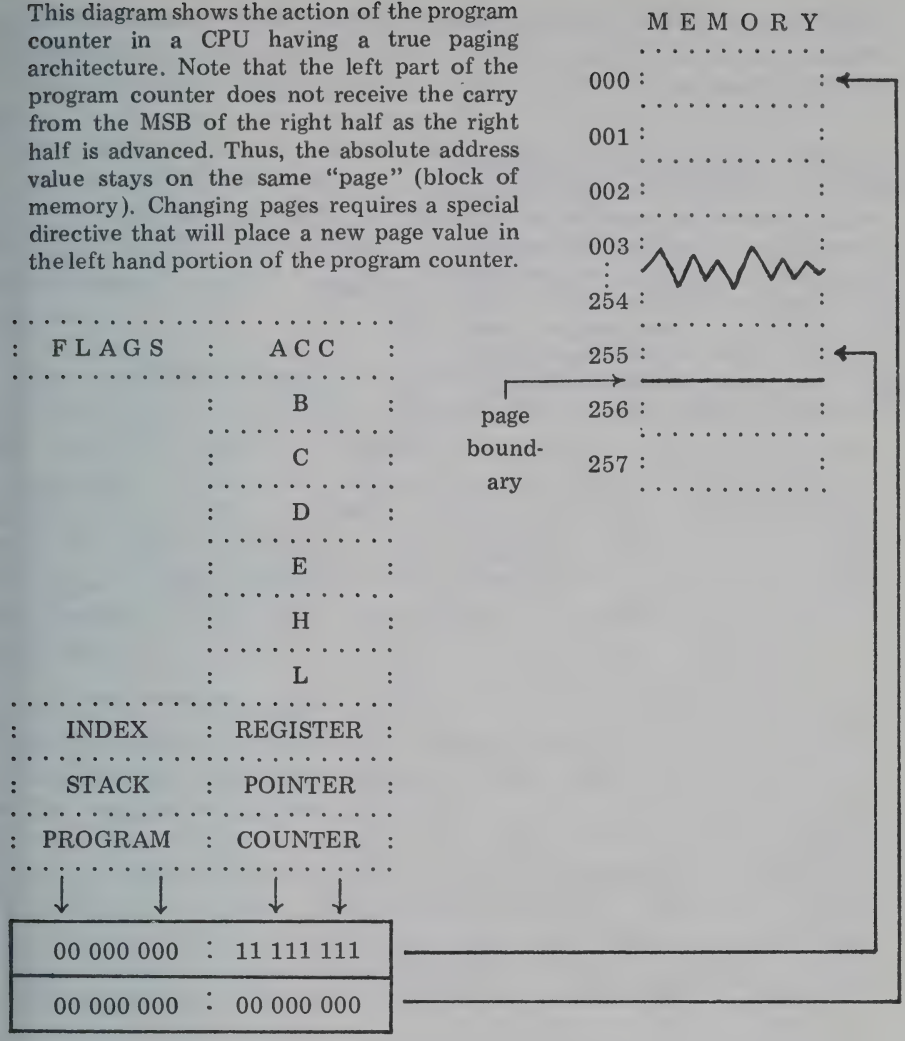
Paging, one may observe that the most significant bit of the right half of the sixteen-bit wide program counter is linked to the least significant bit of the left hand portion of the register. A carry is coupled between the two eight-bit registers at the mid-point. The diagram

further illustrates that when the left hand register contains all zeros, and the right hand register contains all ones, the program counter effectively represents an absolute memory address of 255 (decimal). When that value is incremented by one, the left hand eight bits represent the value one, and the right hand register contains the value zero. However, when read as a value in one continuous sixteen-bit register, it represents an absolute address of 256 (decimal).

The next diagram illustrates the situation in a processor incorporating true paging architecture. By viewing the diagram, the reader may observe that the two eight-bit registers which form the sixteen-bit wide program counter are not linked in operation. Thus, there is no carry from the most significant bit position of the right hand eight bits to the least significant bit position of the left hand eight bits. When the value 1 1 1 1 1 1 1 1 binary is in the right hand eight bits, and the value zero is in the left hand eight bits, the program counter represents an absolute address (as in the Pseudo-Paging machine) of 255 (decimal). However, in a true paging machine, when the right hand register is incremented as in the example, it will go to a value of zero, and there will be NO carry into the left hand portion of the program counter. Thus, instead of the program counter representing the absolute address 256, it will represent the absolute address 0! Comparing the two diagrams, the reader may observe that the absolute address locations pointed to by the program counter are considerably different.

How would one get to the absolute address 256 (decimal) in a true paged machine? One would have to deliberately set the desired PAGE value (in this case, 0 0 0 0 0 0 0 1 binary) into the left hand eight-bit portion of the sixteen-bit wide register. This may seem like extra work, and, indeed, it is. A discussion of the reasons for designing microprocessors with architectures that utilize true paging techniques is beyond the intent of this publication. It will be pointed out that for reasons of programming efficiency, most microprocessors that utilize true paging have large "pages" - typically in the range of 1K to 8K words. This is done so that programmers do not have to constantly be concerned with the ramifications of crossing page boundaries which are those points at which one must change the value of the PAGE section of the program counter.

This diagram shows the action of the program counter in a CPU having a true paging architecture. Note that the left part of the program counter does not receive the carry from the MSB of the right half as the right half is advanced. Thus, the absolute address value stays on the same "page" (block of memory). Changing pages requires a special directive that will place a new page value in the left hand portion of the program counter.

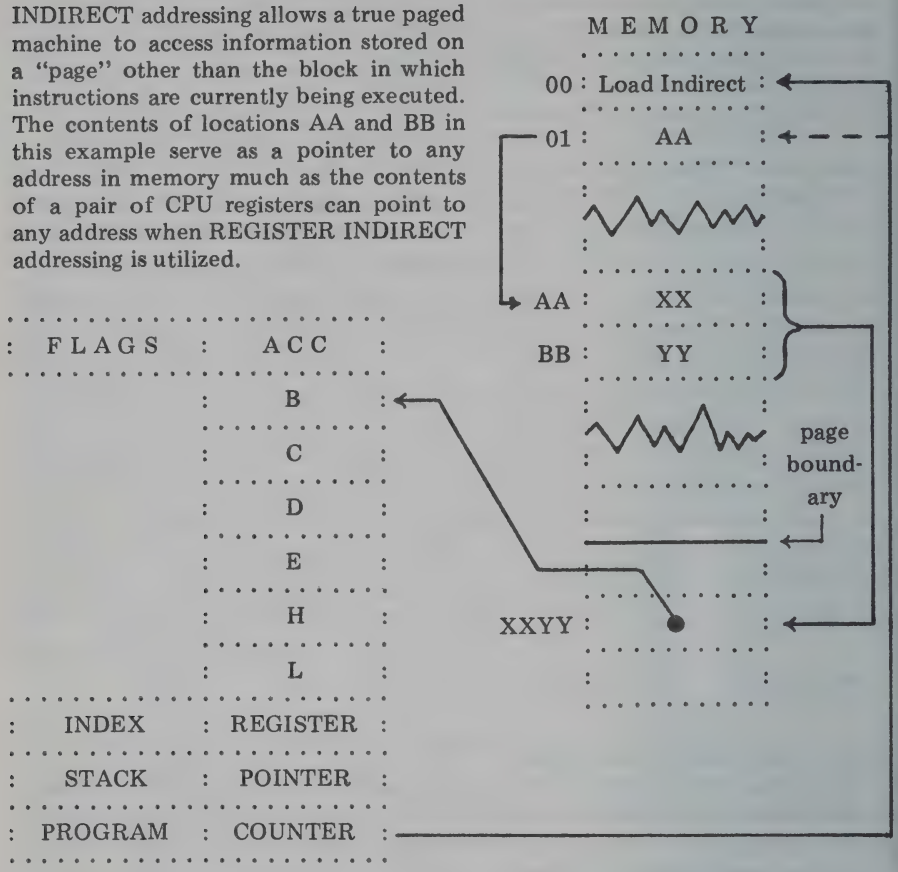


Program Counter Operation in a True Paging Machine

If true paging techniques are rarely encountered in microprocessor systems, then why bring up the subject? The reason is because those relatively few machines that do utilize that type of architecture are virtually required to have the capability of a special addressing mode,

referred to as INDIRECT addressing. To leave out this mode of addressing would not be appropriate. The concept of this addressing mode may best be summarized by the accompanying diagram.

INDIRECT addressing allows a true paged machine to access information stored on a "page" other than the block in which instructions are currently being executed. The contents of locations AA and BB in this example serve as a pointer to any address in memory much as the contents of a pair of CPU registers can point to any address when REGISTER INDIRECT addressing is utilized.



Indirect Addressing in a Paged Computer

INDIRECT addressing is the technique of storing the absolute address of a DATA location in a specific location in memory. In order to access the actual DATA, the processor must first obtain the

absolute address using one of the other types of addressing modes previously discussed. In the example, the two byte LOAD instruction utilizes RELATIVE addressing to specify the starting address of two bytes in memory that contain an absolute address where the actual DATA is stored. The computer must obtain the address stored in those two locations (the locations labeled AA and BB) that contain the address (symbolized as XXYY) and use that value as the effective address value in order to obtain the DATA that is to be loaded into the CPU register (designated as register B in the example).

Notice that the address XXYY is located on the diagram in a section of memory that is on a different PAGE than the address AA. The use of INDIRECT addressing in a true paged machine permits data stored in one section of memory (PAGE) to be accessed by a program or programs residing in other PAGES of memory. A computer that does not utilize a paged addressing architecture does not require the use of INDIRECT addressing for such applications because any location in memory may be reached using DIRECT addressing.

INDIRECT addressing is similar in concept to REGISTER INDIRECT addressing, with the difference being that the MEMORY POINTER registers, when using INDIRECT addressing, are replaced by the memory locations represented in the diagram as starting at address AA. In other words, the MEMORY POINTER actually resides in memory when INDIRECT addressing is used, instead of the CPU registers as when REGISTER INDIRECT addressing is being used. The reader may want to refer back to the diagram earlier in this section that illustrated REGISTER INDIRECT addressing for comparison purposes.

The reader may now be able to see that the number of unique machine codes required to completely specify the instruction set of a microprocessor is a function of not only the number of different types of instructions, and the number of available CPU registers, but also of the number of different addressing modes that the machine is capable of utilizing. Thus, for example, the LOAD instructions which require 36 different commands just to represent transfers amongst the CPU registers using the INHERENT addressing mode are, in-

creased if one provides capability for LOAD instructions that use REGISTER INDIRECT addressing, IMMEDIATE addressing, RELATIVE addressing, and so forth. One may observe that the number of unique instructions (all of the same type) increases rapidly as a function of the different parameters mentioned. Thus, a CPU might actually require 100 or more different machine codes in order to represent all the possible variations of one single class of instructions. The novice, upon reading a boastful manufacturer's specification sheet, might easily be impressed - and then overwhelmed - over the prospects of having to learn all the different variations. However, when beginners realize that these variations are all carefully and logically organized into subsets of a single category, they hopefully will become much more confident of their ability to learn how the machine operates.

With these concepts in mind, the reader can now proceed to learn about the other common classes of instructions found in microprocessors with the realization that there are numerous variations of these types of instructions. The variations are related to what type of addressing mode is being utilized, as well as what specific CPU registers will be affected by the operations.

The next class of instructions to be presented will be referred to as the ARITHMETIC group. One of the major attributes of a microprocessor is the fact that it is able to perform arithmetic operations. However, many readers may be surprised to learn that most microprocessors are only able to execute a few simple arithmetic operations in the sense that most people think of such functions. Most microprocessors have only the capability of adding and subtracting numbers. All other mathematical capabilities are derived from utilizing these basic arithmetic capabilities, and combining them with other types of instructions of a nature that will be presented further on in this chapter. Most CPU's do NOT have the capability of multiplying or dividing. Such capabilities in a microprocessor system are provided by developing sequences of instructions that effectively result in multiplication (or division) being performed.

Most arithmetic operations take place between the various CPU registers and the accumulator(s), or between the contents of words in memory and the accumulator(s). However, some of the more ad-

vanced microprocessors will allow arithmetic operations to occur between various types of CPU registers without affecting the contents of the accumulator(s).

For initial discussion purposes, the basic mathematical operation of addition will be described as it most often occurs. Mathematical instructions that perform ADDITION simply operate to add the current contents of the accumulator register to the current value of the specified CPU register or location in memory, and leave the result in the accumulator. Normally, the value of the register or location serving as an operand is unchanged. However, as a special case, most processors allow the accumulator to add its own contents to itself, thereby providing an instruction which is able to conveniently double a value.

Mathematical instructions generally affect the status of many of the CPU FLAGS. For instance, if there is an overflow condition as a result of the addition operation, the C (CARRY) flag is set to a 1. If there is no overflow as a result of the addition (from the most significant bit in the accumulator), the CARRY flag is reset to the zero state. Similarly, other types of flags, such as the ZERO, SIGN, and PARITY flags, are affected by the results of the addition operation.

Many CPU's provide two fundamental types of ADDITION operations. One type is designed for the addition of binary numbers whose total value can be represented in one byte. The other is designed to facilitate what is known as multi-byte, or multiple-precision, arithmetic. This type of arithmetic must be used to store large numbers that can not be represented in a single register.

The first type of ADDITION operation is illustrated in the diagram below. This operation ignores the initial value of the CARRY flag at the start of the addition operation. The second type of ADDITION directive utilizes the initial value of the CARRY flag and adds it into the least significant bit position at the beginning of the addition operation. Both types of ADDITION instructions set the CARRY flag if the result of the addition causes an overflow condition to occur in the result register, or reset the CARRY if an overflow condition does not occur.

0 0 0 0 0 1 0 1	Accumulator
	+
0 0 0 0 0 0 1 0	B
	=
0 0 0 0 0 1 1 1	Accumulator + B

ADDITION

The diagram below illustrates the result of executing an ADDITION instruction that utilizes the initial status of the CARRY flag. Remember, this type of ADDITION instruction is intended for enhancing a microprocessor's capability to manipulate multi-byte numbers.

1	Carry bit
	+
0 0 0 0 0 1 0 1	Accumulator
	+
0 0 0 0 0 0 1 0	B
	=
0 0 0 0 1 0 0 0	Carry + ACC + B

ADDITION WITH CARRY STATUS REFLECTED

As previously mentioned, the majority of ADDITION commands take place between an accumulator register and another register or location in memory, with the result being left in the accumulator. The examples mentioned refer to the addition of single-byte registers. Some machines do provide for automatic double-byte addition of numbers between selected registers. For example, there may be an instruction that permits the contents of two data registers (such as D and E) to be added to a second pair of registers (such as H and L), with the result of the addition being left in one of the pairs of registers. When this double-register type of addition is provided, the CPU automatically takes care of passing any carries between the most

significant bit of the least significant byte, and the least significant bit of the most significant byte within the register pair. The status of the CARRY flag after such an operation will generally be dependent upon whether or not an overflow condition existed in the pair of registers that held the result of the addition. An accompanying diagram illustrates the execution of such a double-register ADDITION instruction. It should be noted that the double-register ADDITION instruction is very valuable when it provides capability for special registers, such as INDEX or STACK POINTER registers, to have their contents modified by summing them with the contents of other registers.

```
D  01 000 100  11 111 111  E

H  00 111 000  10 000 000  L

H  01 111 101  01 111 111  L
```

DOUBLE REGISTER ADDITION

There is a special type of instruction related to the ADDITION instruction which is provided in some microprocessors. This type of instruction is referred to as the DECIMAL ADJUST operation. The execution of this type of instruction causes the binary contents of a register to be converted to two four-bit groups. Each group represents a binary coded decimal digit. The DECIMAL ADJUST instruction operates on a single register in the manner described next, with the results as illustrated in the accompanying diagram.

Essentially, the DECIMAL ADJUST directive adds a count of six (decimal) to the right or left group of four bits within an eight-bit register (while taking account of, and setting or resetting as required, the status of the DECIMAL CARRY and CARRY flags) whenever the value in the right or left four-bit groups is in the range 10 (decimal = 1010) to 15 (decimal = 1111). Registers containing BCD encoded numbers can be maintained (and this may be extended to

apply to numbers consisting of many digits stored in multi-register format) throughout a program by the use of the DECIMAL ADJUST instruction, instead of having to deal with alternative methods that convert decimal numbers (used by most people) to binary numbers (used by the computer) and vice versa. The inclusion of the DECIMAL ADJUST instruction in a microprocessor's instruction set can be a valuable asset to those interested in performing certain types of mathematical operations (particularly if the applications involve accepting information from BCD encoded instruments, or sending data to instruments or displays that use the BCD encoding technique).

```
                 1
0   1001 1011        Step 1

                 1
0   1010 0001        Step 2

                 0
1   0000 0001        Step 3
```

DECIMAL ADJUST OPERATION

The second type of mathematical instruction that most microprocessors are equipped with is the SUBTRACT directive. Actually, though the SUBTRACT instruction is provided in most CPU devices, it is really provided as a matter of convenience rather than a necessity, because the microprocessor actually performs subtraction via an addition process. What is meant by this? The following discussion and diagrams will help clarify the statement.

In a digital computer having fixed length registers, such as registers that contain eight binary bits, it is customary to deal with positive and negative numbers using a convention known as TWO'S COMPLEMENT arithmetic. This convention calls for the most significant bit in a register (or the most significant bit in the most significant byte of a multi-byte structured register) to have a value 0 if the number

represented by the remaining bits is zero or positive in value. If the number represented in the register (or registers) is to be considered as a negative number, the most significant bit is a 1. Furthermore, the convention calls for a negative number to be represented as the complement of its equivalent positive number plus a count of one. The diagram below illustrates how the number five and its two's complement value minus five would be represented using this convention. The diagram illustrates how the number minus five may be represented by complementing (that is, changing all the bits that are a zero to a one, and vice versa) the value plus five and then adding a count of one to the least significant bit position of the complemented value. The result is known as the two's complement of five (+5) which is minus five (-5).

0 0 0 0 0 1 0 1	Initial value
1 1 1 1 1 0 1 0	Complemented
1	+ 1
	=
1 1 1 1 1 0 1 1	Negative value

TWO'S COMPLEMENT OPERATION

What is so great about two's complement arithmetic? Notice in the next diagram that if the value plus five and its two's complement representation minus five are added together, the result is precisely what one would expect. It is zero. With numbers represented using two's complement notation, the computer can effectively subtract by adding! For instance, to subtract five from seven, the computer could actually proceed by taking the two's complement of five (that is, complementing the individual bits of the binary representation that is equal to five, and then adding one to that complemented value), and then adding the result (-5) to the number seven. The result of this addition would be the value two - exactly what one should obtain when five is subtracted from seven. Most microprocessors actually perform subtraction by going through the process of obtaining the two's complement value and then adding.

00 000 101	+5
	+
11 111 011	-5
	=
00 000 000	Zero

TWO'S COMPLEMENT ADDITION OPERATION

Computers that provide SUBTRACT instructions for the programmer's convenience typically perform the two's complement process when executing such an instruction. What is important for the reader to realize is that if a particular CPU does not provide an actual SUBTRACT type of instruction, the equivalent command is easily obtained by going through the process just indicated. Virtually all computers provide the necessary capability to perform two's complement manipulation. (That is, they have an instruction capable of complementing the bits in a register, and the capability to add the value one to a register.)

As in the case of ADDITION instructions, the SUBTRACTION instructions available in a particular microprocessor may provide several types of capabilities, such as the capability to utilize the contents of the CARRY flag at the start of a subtraction operation for use in multi-byte calculations. SUBTRACTION instructions are generally permitted between the accumulator(s) and other CPU registers or registers in memory with the results being left in the accumulator. Machines that permit multi-register ADDITION instructions are not as likely to include similar capabilities when it comes to subtraction. (This is really because such instructions are a matter of convenience and not so much of necessity. Therefore, their inclusion has relatively low priority in consideration with other types of directives that the CPU designer might like to include as part of an instruction set.)

Many microprocessors provide a type of instruction that is really a special case of a subtraction operation. This special case effectively results in a subtraction operation being performed with the value of a specified register being subtracted from a value in an accumulator,

except that the value of the accumulator (or the operand register) is not actually altered by the subtraction operation. In other words, the subtraction operation is "virtual" in that it takes place within internal CPU areas, and is not reflected in an actual CPU register. However, this type of operation, which is referred to as a COMPARE directive, does result in the CPU flags, such as the ZERO and SIGN

00 000 101	Accumulator
	minus
00 000 010	a register
	equals
00 000 011	Result in a 'virtual' register

Operation of a COMPARE Directive

flags, being set in the same manner as though an actual subtraction operation had occurred between the registers involved. Thus, by subsequently testing the status of the various flags after a COMPARE instruction has been executed, it is possible to determine whether the compare directive revealed a match or non-match condition amongst the registers being tested. In the case of a non-match, using the FLAG results, it is possible to determine if a particular register contained a value greater than or less than the value in an accumulator. This type of an instruction thus allows a programmer to alter the operation of a computer based on the contents of a CPU register or memory location at the time a sequence of instructions is actually executed without altering the actual contents of the registers involved in the COMPARE operation.

(It might be noted that the same types of information could be obtained by performing a subtraction operation. Such an operation would typically result in the contents of an accumulator being altered. This may not be desirable in some instances. The inclusion of a COMPARE type of instruction in a computer's instruction repertoire does, therefore, tend to ease the task of the programmer. The same type of result, however, could be obtained in a system that does

not provide a COMPARE instruction through the use of a SUBTRACTION directive in all respects, if the programmer merely saved the contents of the accumulator and used a SUBTRACTION directive. Remember, the only difference between a SUBTRACTION directive and a COMPARE directive is that the latter affects the status of the FLAGS without changing the contents of the accumulator. The former affects the status of the FLAGS while leaving the actual result of the subtraction operation in the accumulator.)

The reader will be reminded here that as with the LOAD and similar types of transfer instructions, there may be a large number of unique mathematical instructions derived from the basic types just presented through the use of different addressing modes. With just the three types of mathematical directives discussed, it is easily possible for a particular CPU to have one hundred or more instructions defined by individual machine code patterns in order to allow the operations to be performed between all of the individual CPU registers and the accumulator(s), or locations in memory and the accumulator(s).

Virtually all microprocessors available on the market provide for the execution of several groups of instructions that are referred to as BOOLEAN LOGIC operations. These operations are typically performed between the contents of a CPU register, or location in memory, and an accumulator. These BOOLEAN LOGIC operations are valuable in a number of applications. In fact, these types of instructions allow a microprocessor to duplicate the type of logic commonly encountered in digital equipment. These logic operations that may be performed by the microcomputer typically include the LOGICAL AND, the LOGICAL OR, the EXCLUSIVE OR, and the COMPLEMENT operations. Each type of logic operation is normally performed on a bit-by-bit basis between the accumulator and the CPU register or memory location specified by the instruction. A detailed explanation of each type of logic operation, and an appropriate example will be presented below. The execution of these types of instructions generally affects the status of the CPU FLAGS. Thus, after performing these types of logic operations, it is possible for the computer to modify its behavior based on the results of a logic operation.

The execution of a Boolean AND instruction by a microprocessor causes each bit of an accumulator register to be compared with the corresponding bit in a register or memory location specified by the instruction. As each bit is compared, a logic result will be placed in the accumulator. The logic result for an AND operation is determined as follows. If both the bit in the accumulator and the bit in the register with which the operation is being performed are a logic one, the accumulator will be left in the logic one condition. For all other possible combinations (the bits are opposite in value, or both bits equal zero), the corresponding bit in the accumulator will be cleared to the zero condition. The example below illustrates the results of a logical AND operation.

```
1 0 1 0 1 0 1 0      Starting contents of the ACC.

1 1 0 0 1 1 0 0      Contents of the operand register.
-------------
1 0 0 0 1 0 0 0      Logical AND result in the ACC.
```

The execution of a Boolean OR operation is performed on a bit-by-bit basis between the accumulator and the contents of a specified CPU register or location in memory. The logical OR operation will result in the accumulator having a bit set to a logic one if either that bit in the accumulator, or the corresponding bit in the operand register is a logic one. The case where both the accumulator bit and operand bit are a one also satisfies the criteria. If neither register has a logic one in a bit position, the accumulator bit for that position would remain in the zero state. The following example illustrates the result of a logical OR operation being performed.

```
1 0 1 0 1 0 1 0      Starting contents of the ACC.

1 1 0 0 1 1 0 0      Contents of the operand register.
-------------
1 1 1 0 1 1 1 0      Logical OR result in the ACC.
```

A variation of the logical OR operation is termed the EXCLUSIVE OR logical operation. This operation is similar to the OR operation just described, except that when the corresponding bits in both the accumulator and the operand register are a one, the accumulator bit will be cleared to the zero condition. Thus, the accumulator bit will be a one after the operation only if just one of the registers (accumulator register or operand register) had a one in the bit position. The example below illustrates the operation.

```
10 101 010     Starting contents of the ACC.

11 001 100     Contents of the operand register.
-------------
01 100 110     Exclusive-OR result left in ACC.
```

Finally, the logical COMPLEMENT operation is a simple directive that allows the contents of a specified register to be complemented. That is, all the bits within a register that are in the logic one condition are switched to the logic zero condition, and all the bits in a logic zero condition are switched to a logic one. This is essentially the same as the logical INVERT operation commonly used in digital circuits. (In some microprocessors, the COMPLEMENT operation can only be performed by an accumulator. Therefore, if the contents of some other register must be complemented, it must be first transferred to the accumulator, the operation performed, and the desired results sent back to the intended register. Note though, that the COMPLEMENT operation does not require a so-called "operand" register.)

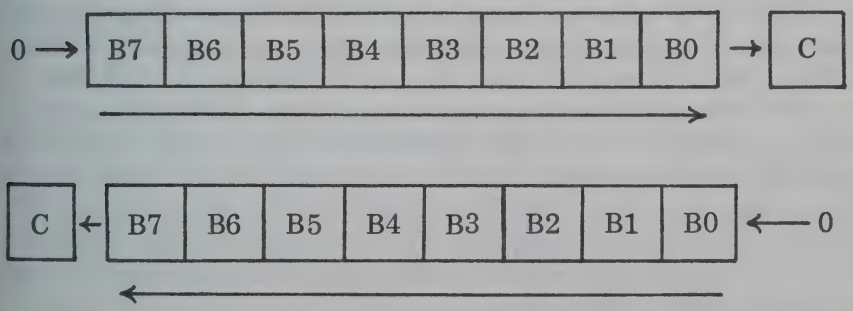
```
10 101 010     Starting contents of the ACC.
-------------
01 010 101     Result of COMPLEMENT
```

As in the cases of all the other types of operations discussed, the number of permutations of the LOGICAL instructions that may be found in a particular microprocessor will be a function of the num-

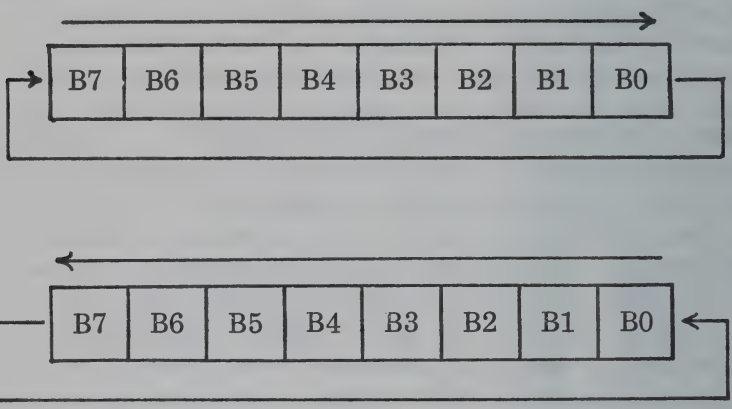
ber of CPU registers available, and the types of addressing modes possible on that particular device.

It is often desirable to be able to shift the contents of a register either right or left. In most microprocessors, only the accumulator register is provided with this capability. Thus, in order to perform a shift operation on the contents of any other CPU register or location in memory, it is necessary to first transfer the data to an accumulator register. The ability to shift data has many uses, as readers who may have had digital circuit experience well know. The concept of shifting data in an accumulator was presented in a previous chapter.

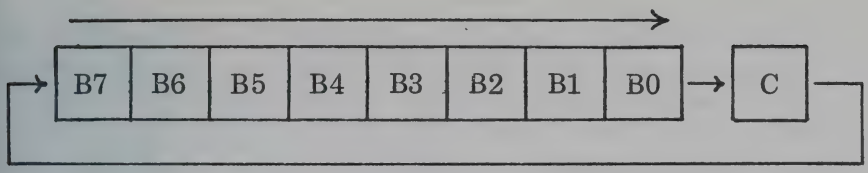
In a fixed length register, a simple shift operation would result in some information being lost because what was in the most significant or least significant bit (depending on which direction the shift occurred) would be shifted completely out of the register! Actually, in some cases this may be desirable, and some microprocessors provide a class of instructions referred to as SHIFT instructions. In these microprocessors, the execution of such an instruction, say for example, a shift to the right instruction, results in the contents of each bit within the register being shifted one bit to the right. The least significant bit in the register is effectively shifted right out of the register. Since for such an instruction there is no bit available for shifting into the most significant bit of the register, a zero state is typically loaded into that position on a RIGHT SHIFT operation. Going in the opposite direction, a LEFT SHIFT operation results in the contents of the most significant bit position being shifted out of the register, and the least significant bit position receives the value zero. The diagram below illustrates the operations.

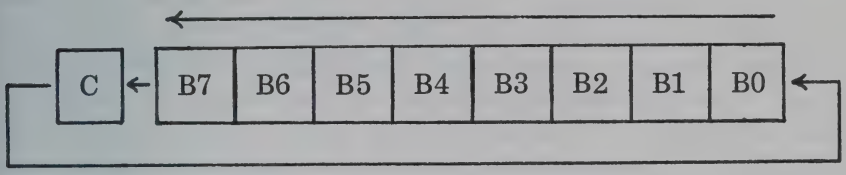


In many cases, however, it is desirable not to lose the contents of a register as it is shifted. That is, it is desirable to move specific bits into different positions within a register while still maintaining all of the data for eventual restoration to original bit positions. Therefore, an operation known as ROTATING is utilized. Instead of just shifting a bit off the end of a register, the bit is brought around to the other end of the register. For example, if the register is rotated to the right, the least significant bit (LSB) will be brought around to the position of the most significant bit (MSB) which would have been vacated by the shifting of its original contents to the right. Or, in the case of a shifting to the left, the MSB would be brought around to the position of the LSB.



Sometimes it is desirable to consider the CARRY flag as an extension of an accumulator. For such cases, microprocessors are often equipped to provide a special set of ROTATE instructions that include the CARRY flag in such operations. In such cases, a ROTATE LEFT directive would result in the MSB being shifted into the CARRY bit position, while the contents of the CARRY bit were circulated around to the LSB position of the register. A ROTATE RIGHT directive would result in the contents of the LSB position being sent into the CARRY bit, while the contents of the CARRY bit were transferred to the MSB. Most machines provide for the two types of operations known as ROTATE with CARRY and ROTATE without CARRY instructions.





It might be noted that rotate and shift operations are particularly valuable for certain types of mathematical operations such as multiplying or dividing. This is because shifting the contents of a register to the left effectively multiplies a binary number by a power of two. Shifting such a number to the right provides the reverse operation of division by powers of two. Rotate and shift operations are also valuable for a number of other considerations, including the transfer of information to and from external devices connected to the microcomputer.

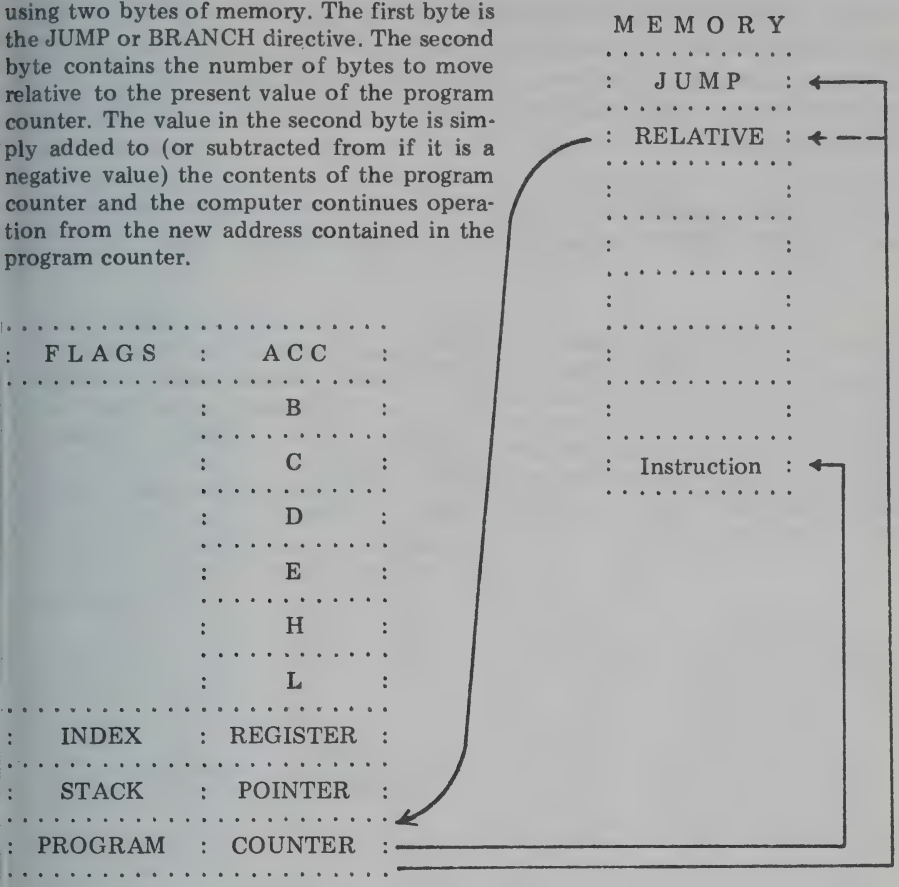
The instructions discussed so far have all been commands that operate on data. The programmer arranges a sequence of these types of instructions in memory. At the proper time, the computer proceeds to execute the instructions in the order in which they are encountered. The computer automatically reads the contents of a memory location, executes the instruction it finds there, and then automatically increments the register known as the PROGRAM COUNTER. The reader has been informed that the PROGRAM COUNTER is incremented as each byte of an instruction sequence is executed so that the computer will know where to obtain the next instruction (or part of an instruction in the case of multi-byte directives). Normally, this routine procedure of incrementing the address contained in the PROGRAM COUNTER each time a byte of an instruction is processed is all that is required to guide the operation of the CPU. Indeed, as has been pointed out to the reader, this simple operation - incrementing the PROGRAM COUNTER - gave

it its name. However, at some point in a computer's operation, it may be necessary to place a specific value into the PROGRAM COUNTER register. One such time would be when the computer is initially started. That is, some value must be placed in the PROGRAM COUNTER register in order to set it to an address value where it will find a sequence of instructions to be executed. Moreover, it is often desirable to be able to perform a series of instructions located in one section of memory, and then skip over a group of memory locations (that might, for instance, contain data) and continue executing another sequence of instructions in a different section of memory. To accomplish this type of behavior on the part of the computer, it is necessary to place a new address value into the PROGRAM COUNTER. Doing so will cause the computer to start executing instructions at the new memory address location placed into the PROGRAM COUNTER. One type of instruction that will perform this function is referred to as the JUMP instruction. This type of directive is also commonly referred to as a BRANCH instruction.

JUMP or BRANCH instructions generally utilize the addressing method previously described as DIRECT addressing. That is, a JUMP instruction requires more than one word of memory in order to be completely specified. The first byte in the multi-byte JUMP instruction contains the machine code for that type of instruction. The second byte will then contain either a relative offset value, which may be added to the value in the PROGRAM COUNTER, or contains half of an absolute address value. In the latter case, a third byte would contain the other half of a typical sixteen-bit absolute address value. (Note that in some machines there is an addressing mode that allows the second byte of a two-byte JUMP instruction to specify an absolute address in the first 256 (decimal) locations in memory.)

In the case where a BRANCH instruction uses only one byte for the address portion, the byte is intended to contain an offset value which is to be added to the current value in the PROGRAM COUNTER. Naturally, when this is done the program counter value will abruptly change to the new address as a result of the addition (instead of the normal increment that occurs as each byte of an instruction is executed). Therefore, the next instruction to be executed

A RELATIVE JUMP or BRANCH directive using two bytes of memory. The first byte is the JUMP or BRANCH directive. The second byte contains the number of bytes to move relative to the present value of the program counter. The value in the second byte is simply added to (or subtracted from if it is a negative value) the contents of the program counter and the computer continues operation from the new address contained in the program counter.



JUMP or BRANCH Instruction Operation Utilizing Relative Addressing

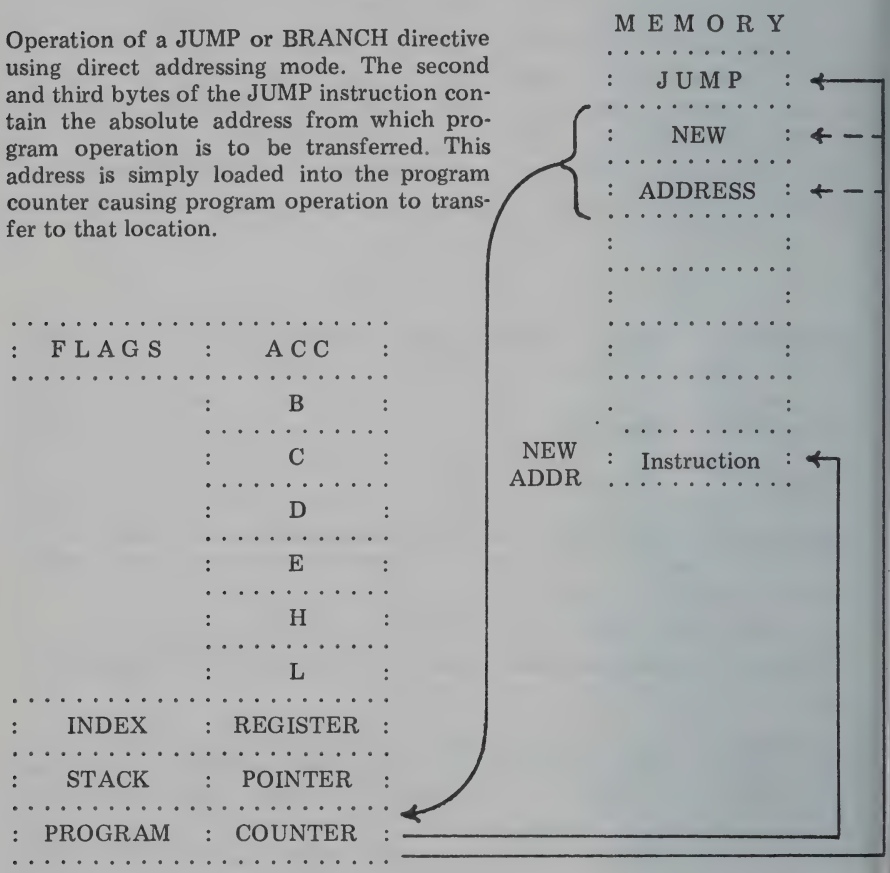
(as indicated by the program counter) would be at a different location in memory than would have been the case had the program counter been routinely incremented (unless the offset value happened to be a count of one).

In the case where a BRANCH instruction is followed by two bytes containing an absolute address value, the contents of the program

counter (when the last byte of the three-byte directive has been processed) will abruptly be changed to the new address. Thus, program execution would be continued from a different section in memory (unless, again, the address given in the BRANCH directive was the address of the next location in memory following the BRANCH instruction itself).

Note that the BRANCH or JUMP instruction does not manipulate any so-called DATA. This type of instruction is only used to alter the

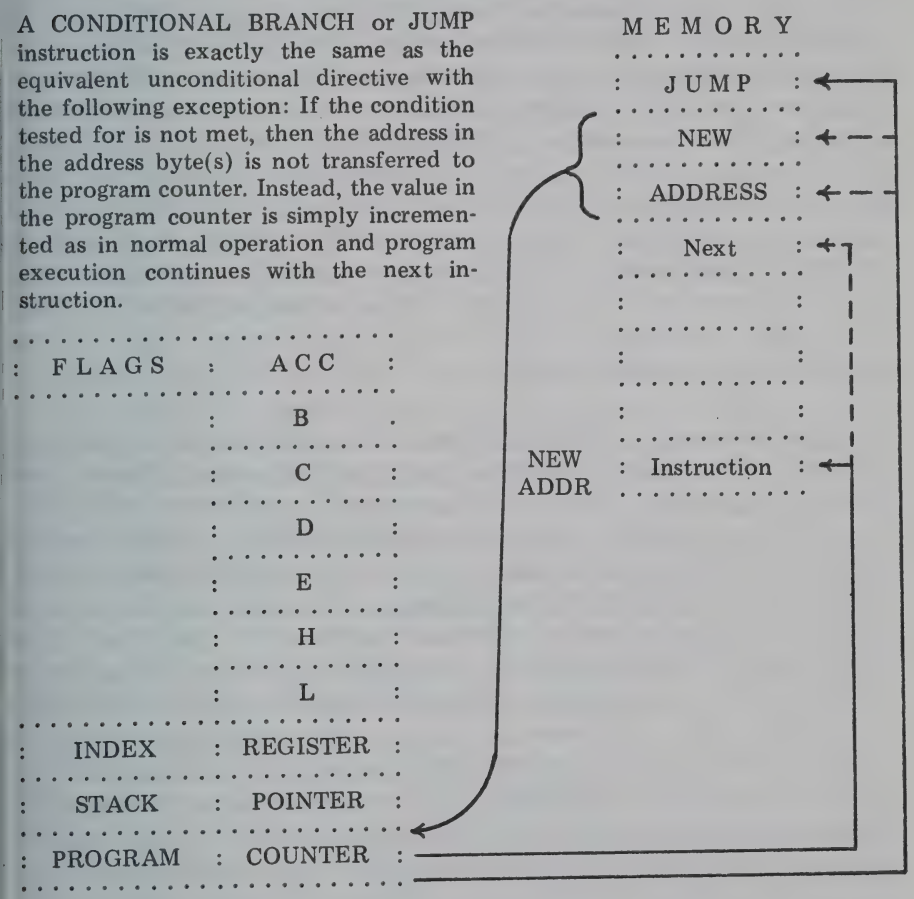
Operation of a JUMP or BRANCH directive using direct addressing mode. The second and third bytes of the JUMP instruction contain the absolute address from which program operation is to be transferred. This address is simply loaded into the program counter causing program operation to transfer to that location.



JUMP or BRANCH Instruction Operation Utilizing Direct Addressing

normal incremental operation of the program counter, and thus gives the computer the capability to jump or skip about to different addresses in memory. This capability allows a microcomputer to move around sections of memory that contain data, or to execute different sequences of instructions. This capability is one of the major reasons why a computer can be so versatile. Through the use of jump instructions, it is possible to have a large variety of different operational sequences stored in a computer's memory, and perform those sequences of operations at will by merely instructing the com-

A CONDITIONAL BRANCH or JUMP instruction is exactly the same as the equivalent unconditional directive with the following exception: If the condition tested for is not met, then the address in the address byte(s) is not transferred to the program counter. Instead, the value in the program counter is simply incremented as in normal operation and program execution continues with the next instruction.



CONDITIONAL JUMP Instruction Operation

puter to jump to the location where a desired sequence of instructions is stored.

Furthermore, not only are microprocessors generally endowed with the simple JUMP or BRANCH capability, but this power is almost always vastly increased by the fact that microprocessors generally have a set of additional JUMP or BRANCH instructions called "conditional" directives. They are called "conditional" because the computer can be directed to test the status of one or more chosen FLAGS (such as the ZERO, SIGN, or CARRY flags). Then, based on the status of a flag (that is, if it is in a desired state), a JUMP or BRANCH will be performed. If it is not in a desired state, the machine will not execute the branching directive, and the computer can proceed to continue to execute instructions in the present sequence. This powerful capability provides a means for the computer to make decisions, and to modify its own operation as a function of the status of the various flags at the precise time that a program is executed. Naturally, the status of these flags may be affected by the results of calculations, or the type of information it may have received from external devices within a system. One may now begin to see how a computer can be endowed with complex decision-making capabilities.

Quite often when using a computer, it is possible that a particular sequence of instructions can be used many times in different portions of a program. This is such a common phenomenon that CPU designers have taken it into account and generally include a special class of instructions to facilitate capitalizing upon this aspect of computer utilization. Rather than having to keep entering the same sequence of instructions at different locations in memory (which would result in a lot of memory being used to perform the same type of function at various points within a program), it is desirable to be able to put the often repeated sequence of commands at just one location in memory. Then, whenever the particular sequence of commands is required by the main portion of the program, it would be advantageous to be able to jump to the sequence of instructions that performed the desired operation, and then return back to the main part of the program. This type of operation is a standard practice when using microcomputers. The frequently used series of commands is designated a SUBROUTINE. The special class of instruc-

tions that allow a programmer to repeatedly use a SUBROUTINE by other portions of the main program is generally referred to as the CALL or JUMP to SUBROUTINE group. This nomenclature is sometimes replaced by reference to the BRANCH to SUBROUTINE directive. The difference between the JUMP to SUBROUTINE type of instruction verses the JUMP directive previously discussed is that the JUMP to SUBROUTINE directive tells the CPU to perform special operations. These operations will allow the computer to return back to the instruction that follows the JUMP to SUBROUTINE directive when it encounters a special instruction at the end of the sequence of commands that make up the SUBROUTINE. This special SUBROUTINE-ending instruction is commonly referred to as a RETURN directive. A great deal of computer power is provided by this class of instructions, particularly because (as in the case of the regular JUMP directive) this class of instructions generally includes a number of CONDITIONAL CALL directives that may be used to route operations to a particular subroutine, or ignore that same subroutine as a result of the status of the CPU flags at the time that a CONDITIONAL CALL directive is actually processed during the execution of a program.

As with the JUMP class of instructions previously described, the CALL instructions typically require two to three words in memory in order to be fully specified. The first word will be the machine code that informs the CPU that a CALL instruction is to be executed. The next byte will then contain the relative offset value that is to be added to the contents of the PROGRAM COUNTER to reach the beginning of the SUBROUTINE. Or, it will contain one half of the absolute address value that is to be placed in the program counter. In that case, DIRECT or EXTENDED addressing would be in use, and a third byte containing the other half of the absolute address would complete the multi-byte directive.

The execution of a CALL or BRANCH to SUBROUTINE instruction is a bit more complicated than a simple JUMP directive. This is because the CPU must save the address that the computer is to return to once the instructions in the subroutine have been executed. What address would this be? It would be the address of the instruction that immediately follows the last byte of the CALL instruction. Simply put, it would be the address value obtained by adding a count

of one to the contents of the PROGRAM COUNTER after it has completed processing all the bytes of the CALL directive (but before it had placed the address value for the subroutine into the program counter).

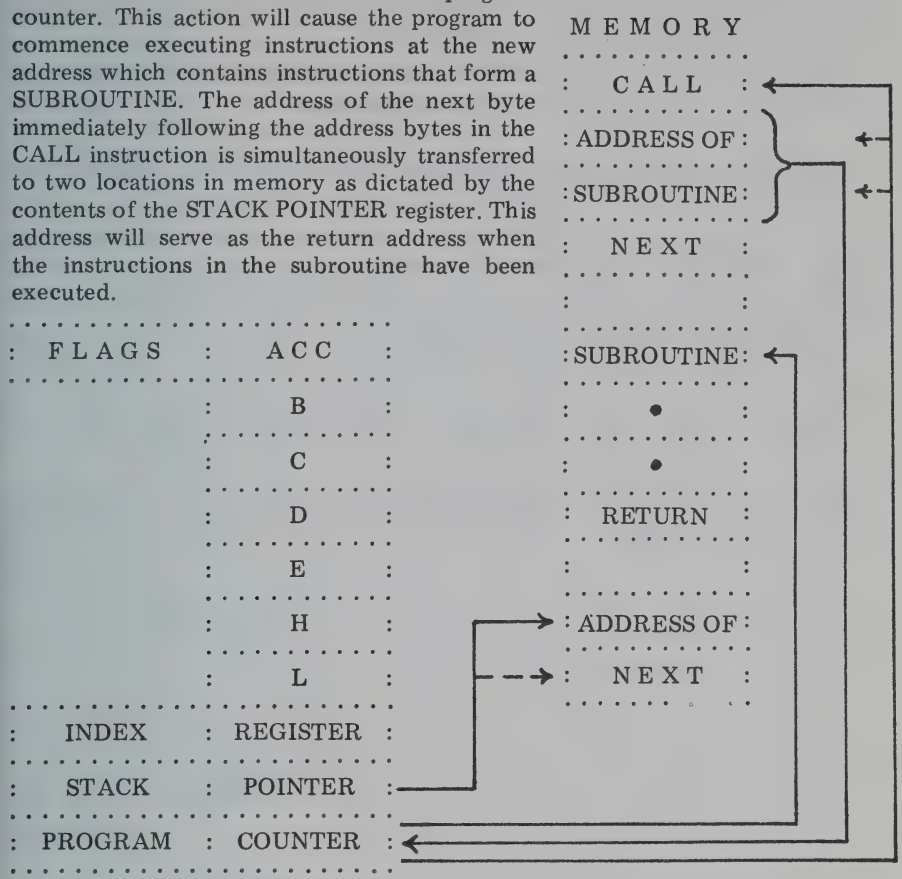
Where could it store the return address (value in the program counter plus one)? In most microcomputers, this address would be stored in two consecutive bytes starting at the address in memory pointed to by the contents of the STACK POINTER. (The address pointed to by the STACK POINTER would be initialized by appropriate directives before a JUMP to SUBROUTINE instruction was encountered in order for the computer to operate properly.)

After this operation has been performed, the address of the start of the SUBROUTINE is placed in the PROGRAM COUNTER, and the computer resumes operation from the new address loaded into the program counter by the CALL directive. (Internal registers within the CPU would be used to temporarily hold the subroutine address while the contents of the program counter plus one were placed in the area in memory pointed to by the value in the STACK POINTER.) The following diagram illustrates the operations that would occur in a typical microcomputer when a CALL instruction was encountered.

It might be noted that as the STACK POINTER was used to store an address value in memory, it would typically automatically be incremented (or decremented) to the next memory location as each byte was processed. In this manner it is possible for a computer to "nest" a number of subroutines one inside another. It is also important to remember that other types of instructions, such as LOAD directives, may typically be used to initialize or alter the value in the STACK POINTER, and thus control the memory areas that will be used when addresses are stored by CALL instructions.

The conclusion of a SUBROUTINE is indicated by terminating a sequence of instructions with a type of directive referred to as a RETURN command. When the CPU encounters a RETURN instruction, it proceeds as follows. It first obtains the address stored in the area of memory pointed to by the STACK POINTER. (Since the STACK POINTER is typically incremented (or decremented) each

When a CALL instruction is encountered, the address contained in the 2'nd & 3'rd bytes of the directive are transferred to the program counter. This action will cause the program to commence executing instructions at the new address which contains instructions that form a SUBROUTINE. The address of the next byte immediately following the address bytes in the CALL instruction is simultaneously transferred to two locations in memory as dictated by the contents of the STACK POINTER register. This address will serve as the return address when the instructions in the subroutine have been executed.



Execution of a CALL Instruction

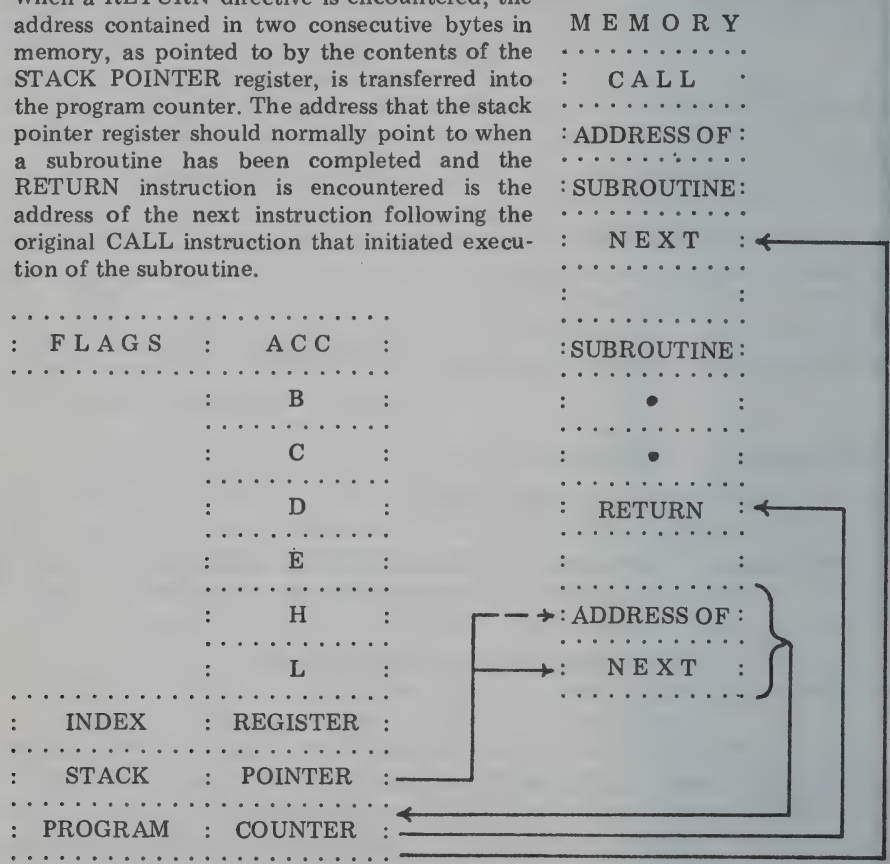
time it is used to store a portion of an address in memory, the reverse procedure, that is, a decrement (or increment) of the STACK POINTER must be made PRIOR to recovering each byte of an address that has been so stored.) The address recovered from the area in memory pointed to by the STACK POINTER is placed as the new contents of the PROGRAM COUNTER. Since the address recovered is the address of the next instruction that followed the original

CALL directive, the computer will then continue operations immediately following the point in the main program from which the SUBROUTINE was originally "called."

The diagram below illustrates the operation that would typically occur upon processing a RETURN instruction.

As is the case with most microprocessors having the capability to

When a RETURN directive is encountered, the address contained in two consecutive bytes in memory, as pointed to by the contents of the STACK POINTER register, is transferred into the program counter. The address that the stack pointer register should normally point to when a subroutine has been completed and the RETURN instruction is encountered is the address of the next instruction following the original CALL instruction that initiated execution of the subroutine.



Execution of a Subroutine-Concluding RETURN Instruction

perform or ignore jumps to subroutines as a function of the condition of various status flags, so most microprocessors provide a similar capability to exit from a subroutine, or to continue operation within a subroutine as a function of the status of various CPU flags. These analogous exit directives are called CONDITIONAL RETURN instructions.

Notice that as with the JUMP class of instructions, the CALL instructions and subroutine terminating RETURN directives do not have anything to do with manipulating DATA. They are used only to alter the contents of the program counter, and, therefore, provide a computer with capability to skip about and execute instruction sequences stored in different sections of memory. Thus, it is possible to divide the use of memory within a computer system into areas that contain instructions that tell the computer what to do, and other areas that contain DATA that the computer manipulates.

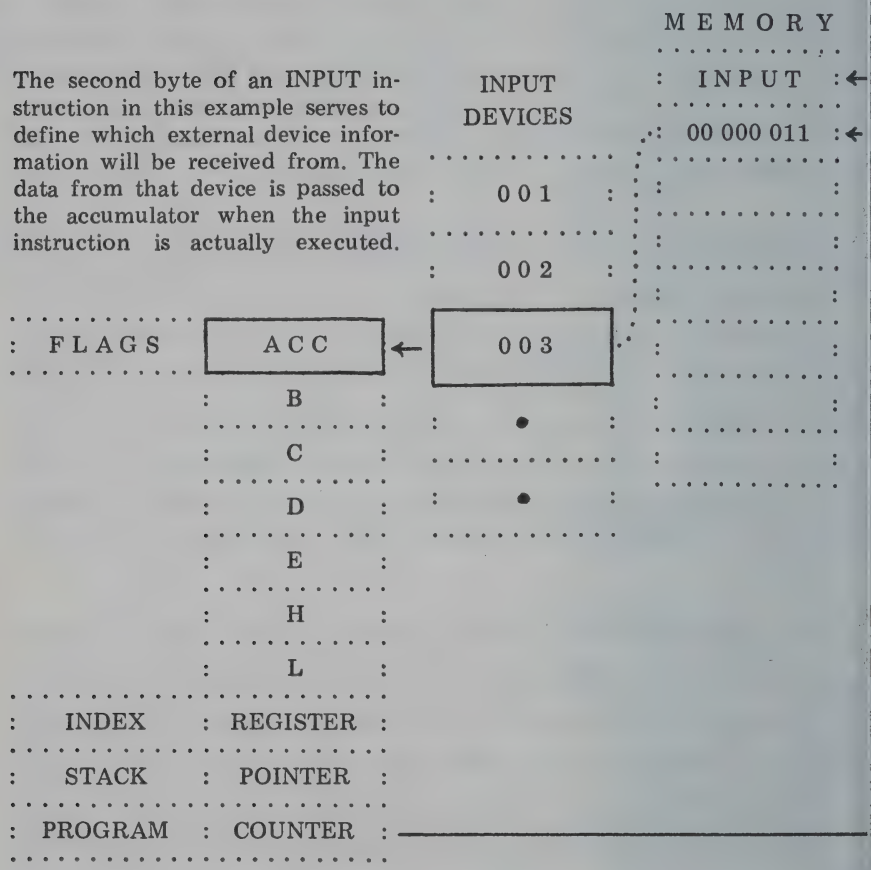
The next class of instructions to be discussed is a very simple class commonly referred to as I/O (Input/Output) instructions. In some microcomputer systems, it is possible to connect external devices to the computer in such a fashion that they emulate a location in memory. That is, each device is assigned one or more memory addresses. In such cases, the CPU will not require a special class of I/O instructions to process data going to or coming from an external device. The computer will simply use the LOAD and STORE group of instructions from and to the external devices using appropriate addressing modes just as though it was manipulating data at a particular location in memory.

Some microcomputer systems, however, do not have I/O devices connected to the computer in such a fashion that they emulate locations in memory, but rather, have them connected through a separate electronic network that is controlled by a special group of I/O directives. Depending on the particular architecture of a CPU, these I/O instructions may require one or two bytes to be fully specified. One byte instructions are generally used if the number of I/O devices that can be connected is small, thus allowing a unique device identifying pattern to be encoded within just a few bits of the input or output instruction byte. In cases where the number of possible devices is large, an additional byte may be used to specify a parti-

cular I/O unit. The next several paragraphs assume a CPU havir capability to communicate with up to 256 INPUT devices, and a equal number of OUTPUT devices.

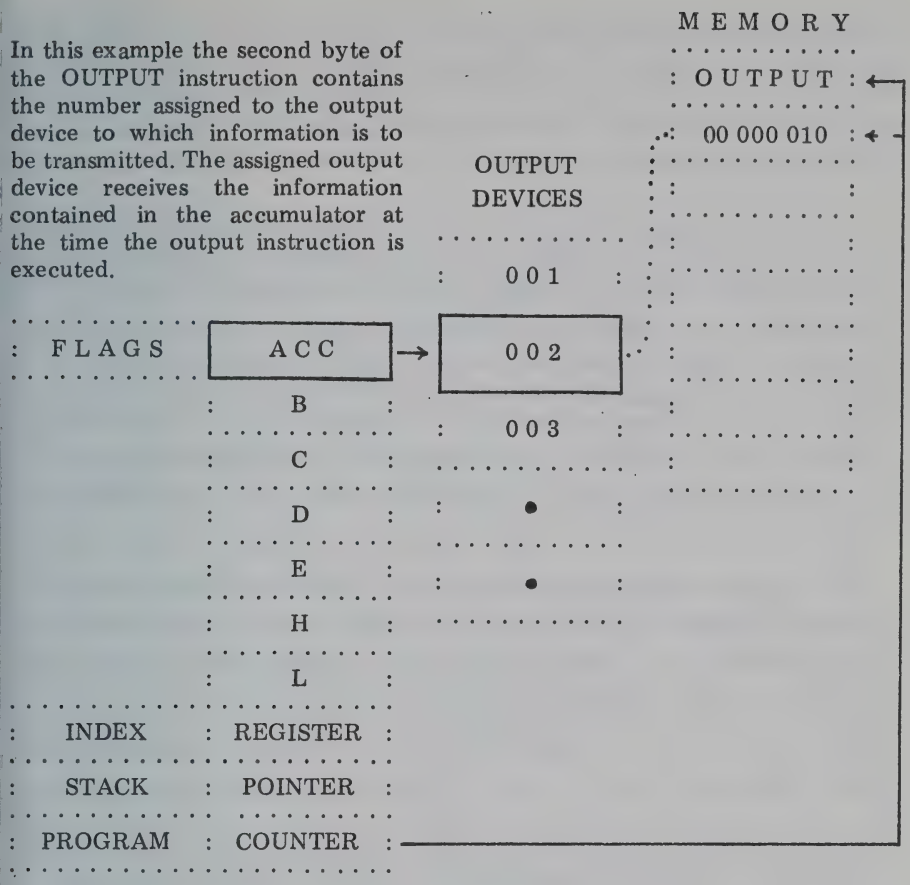
In order to receive information from an external device in such system, the computer must utilize a group of special signal line Each group of signal lines may itself consist of up to eight individu INPUT signal lines. Each line is considered to represent the status c

The second byte of an INPUT instruction in this example serves to define which external device information will be received from. The data from that device is passed to the accumulator when the input instruction is actually executed.



Operation of an INPUT Instruction

In this example the second byte of the OUTPUT instruction contains the number assigned to the output device to which information is to be transmitted. The assigned output device receives the information contained in the accumulator at the time the output instruction is executed.



Operation of an OUTPUT Instruction

one bit in an eight bit register. A group of these signal lines is accepted at the computer through an electronic arrangement commonly referred to as an INPUT PORT. The computer selects an individual INPUT PORT by specifying the port number as the value contained in the second byte of an INPUT instruction. When such an instruction is executed, the computer will examine the group of eight lines coming into the designated INPUT PORT, and transfer the logic states of those lines to the accumulator. Various types of

external equipment, such as an electronic keyboard, or information from an instrument, can thus be transferred through an INPUT PORT to the accumulator. Once in the accumulator, the data can be manipulated by all the other classes of instructions that the CPU is capable of performing.

In order to output information to an external device in such a system, the computer will utilize a separate group of signal lines referred to as OUTPUT PORTS. Once more, each OUTPUT PORT would actually consist of eight individual signal lines, each line being used to carry the logic status from one bit in the accumulator to an OUTPUT device. Through the use of OUTPUT instructions, the computer is able to pass information from the accumulator of the CPU to external devices such as a cathode-ray-tube display device, a teletypewriter, or other types of machinery that are to be placed under control of the computer.

The classes of instructions discussed so far may be considered as the major classes of instructions found on most microprocessors. As the reader may now realize, there may be many variations and permutations of these types of instructions as a function of the different types of addressing modes that may be provided in a particular microcomputer system. Additionally, most microprocessors will have some instructions that do not fall into the major classes that have been discussed. The number and types of such instructions that fall outside the major classes will vary depending on the architecture and complexity of individual CPU devices. Some of the more frequently encountered instructions that fall into this miscellaneous category will be presented next for the reader's enlightenment.

Perhaps one of the most often encountered instructions in the miscellaneous category is the register increment or register decrement directive. As commonly provided, this instruction allows a specified CPU register (and oftentimes a location in memory) to be increased or decreased by a value of one. However, some machines provide capability to increment or decrement a register by a specified value that is indicated in a second byte. Increment/decrement instructions are particularly valuable in microcomputers as they may be used to increase or decrease the value contained in memory pointer registers. Most machines are designed so that the increment and/or decrement

instructions affect the status of the CPU flags upon completion of the directives. Thus, these instructions are also useful for forming what are known as "program loops" as repetitive sequences of operations can be terminated when a certain value (such as zero) is reached in a register that is being incremented or decremented.

A special case of the increment/decrement type of instruction is sometimes supplied and given the nomenclature of an "increment and skip if zero" directive. This type of instruction results in a specified CPU register (and in special machines, specified memory locations) being incremented. Then, if the increment results in the value in the register becoming zero, the next instruction in the sequence being executed will be skipped. That is, effectively jumped over. If the value in a register is not zero after the increment operation, the next instruction is executed. By having the instruction immediately following the "increment and skip if zero" instruction be a JUMP directive, the programmer can arrange a sequence that will cause the computer to branch around one group of instructions when the contents of a register are not zero, and to execute the group of instructions when the value reaches zero in that register. Sometimes, an equivalent "decrement and skip if zero" instruction may be provided along with, or in place of, the "increment and skip if zero" directive just described.

Other types of instructions frequently found in microcomputers are those that will allow the programmer to manipulate the status of particular flags, or individual bits within a register. As an example, quite often an instruction will be provided that will direct the CPU to change the status of the CARRY flag. Similarly, the capability to set the CARRY flag to a one or a zero condition by individual directives is often provided. These types of instructions are particularly useful when performing certain types of operations in that they give the programmer a convenient means to set particular flags (or bit(s) in a register) to a known state prior to beginning a sequence of operations.

Several other types of instructions frequently found in a microcomputer are those that will direct the CPU to HALT or cease opera-

tions, or to perform a NOP "no operation" directive. A "no operation" directive is typically used to consume time during computer controlled operations that require precise timing intervals. (NOPs also turn out to be extremely handy instructions to have around when making minor changes to programs!)

The reader has now been introduced to the major types of instructions that a typical microprocessor CPU is capable of performing. The reader should remember that individual CPU devices may have variations from those discussed. The reader should realize that in order for a microcomputer to perform any useful function, it must be programmed by arranging a suitable sequence of instructions within the memory banks of the microprocessor system. The length of instructional sequences may vary from just a few to many thousands of instructions. The process of learning how to combine individual instructions into instructional sequences that will perform complicated operations is rightly the subject of somewhat more advanced textbooks than this publication. However, in the next chapter, some of the processes and techniques involved will be presented to give the reader an introduction to the subject.

PROGRAMMING - AT THE MACHINE LANGUAGE LEVEL

In the previous chapter the reader learned that the typical microprocessor CPU is able to perform a variety of instructions. The types of instructions were described as consisting of closely related groups. While there are not that many groups, the fact that each instruction may utilize several addressing modes means that a particular group of instructions may contain one hundred or more different directives. Each directive requires a unique binary pattern of ones and zeros in order to be individually identified by the CPU. Add all the instructional groups together, and one usually finds that most of the 256 individual patterns that can be represented in 8 binary bits have been utilized, each to represent a different instruction. This means, simply, that there are typically several hundred different binary patterns (each of which will direct the computer to perform a specific function) that a person planning on controlling the operation of a microprocessor at the machine language level should plan on learning! That is just the beginning of what a machine language programmer must know!

Fortunately, programming at the machine language level is not the only manner in which a computer can be instructed what to do. Machine language programming is so complex that requiring everyone to program a computer using the method would severly restrict effective use of the machine to a relatively small percentage of the population. Fortunately, another method of controlling a computer, referred to as HIGH LEVEL PROGRAMMING, is available to the average computer user. The important aspects of this type of programming will be presented in a following chapter. However, it is important to realize that the high level programming capability is derived from programs constructed using machine language techniques. Additionally, certain types of functions and capabilities can only be obtained by directing the operation of a microprocessor at the machine language level. Hence, it is important for virtually anyone planning on owning a computer to have at least a rudimentary understanding of the aspects of programming at the machine language level.

Just what is machine language programming? It is the process of informing a computer system exactly what to do by arranging the proper sequence of directives in the memory of the computer. It uses the actual binary patterns that represent the directives and the actual binary numbers that represent addresses in memory.

It is extremely exacting work!

A computer is only capable of performing precisely what it is told to do by way of the binary patterns it obtains. If the directives it obtains are faulty or incorrect (in terms of what one really wanted the machine to do), the computer will still perform exactly the directives it decodes. It can not do anything useful unless the binary patterns representing directives are arranged in a logical sequence that will guide it to reach the desired objective. When programming a computer at the machine language level, one cannot just be 99 percent accurate when telling the machine what to do. One must be 100 percent correct. This demand on the part of the machine for absolute accuracy places a burden on human programmers (who have a tendency to make an occasional error). This burden rapidly drives some people to abandon attempts to program computers at the machine language level!

This chapter will present the reader with an overall view of machine language programming. Various aspects about what is required of the programmer will be discussed. The fundamental methods of developing programs using this technique will be presented. Some simple examples will be illustrated. Some of the practical considerations for using this level of programming will be discussed. The purpose of the chapter is to simply acquaint the beginner with the most fundamental way in which a microcomputer is controlled, and to illustrate what is required on the part of the machine's boss in order to direct its operation at this most direct level!

What is a computer program? It is simply a sequence of instructions arranged in memory so that when each instruction is executed in order, a specific task will be accomplished. A program may consist of just a few instructions, or it may be made up of many thousands of instructions. Programs that consist of relatively few instructions (all of which are related to just one specific type of task) are often

referred to as routines or subroutines. These routines may then be tied in with other routines to form major programs. Remember, however, that the distinction between a program and a routine is essentially one of semantics with the connotation that routines are sections of a larger structure.

Another name for a program is an ALGORITHM. An algorithm is a set of rules or procedures by which one may proceed to solve a problem or perform a particular operation. For instance, an algorithm to obtain water from a faucet is to "turn the handle clockwise until water flows." An algorithm to find the average of a group of numbers is to "add the individual numbers together to obtain a total, and then divide the total by the number of individual values."

It has been said that a computer is capable of performing any operation for which a clearly defined algorithm has been determined. From a practical standpoint, one can arrange systems so that a computer can perform the types of functions for which algorithms were described in the previous paragraph. Thus, one could connect a clockwise rotating motor to a switch controlled by a microprocessor. One could also place a moisture sensing instrument under a water spigot that would close a switch when it became wet. One could connect the contacts of the switch to a microcomputer. One could then arrange a series of instructions in the computer's memory that would direct the following series of operations:

1. Turn the clockwise rotating motor connected to the water faucet on (by activating a switch controlled by the computer).

2. Test to see if the switch from the moisture sensing device has closed.

3. If not, repeat step 2 again.

4. If the switch has closed, turn off the motor connected to the water faucet (by deactivating the motor switch).

Similarly, a computer may be used to find the average of a series of numbers by following a series of directives such as:

1. Place the series of numbers to be averaged as "data" in an area in memory reserved for that purpose.

2. Add the numbers to obtain a total.

3. Determine how many numbers were in the series (count how many numbers).

4. Divide the total of the numbers by the number of numbers (using a division algorithm, since most microprocessors can not directly use a "division" instruction).

However, the statement about a computer being able to perform any function for which an algorithm has been defined is often misconstrued by laymen who think computers can be "told" to do anything. This misunderstanding comes about because they do not realize that there are many types of functions for which no one has been able to define a valid algorithm (and for which there does not appear to be any solution in the forseeable future)! For instance, giving a computer the power to think as people are able to do does not appear possible because it has not been determined how humans acquire the ability. Hence, no valid algorithms exist for duplicating the process, and there is little evidence to suggest that any such algorithm will be found in the less than distant future.

Alas, computer owners will have to be content for some time with having their computer systems perform tasks that are capable of being well defined and converted to a series of steps that can be executed by following the proper directives in a microcomputer's repertoire of commands. The first task in programming a computer is to determine just what task is to be accomplished, and defining the proper algorithm by which it is to accomplish the task. This statement may sound a lot simpler than it sometimes turns out to be!

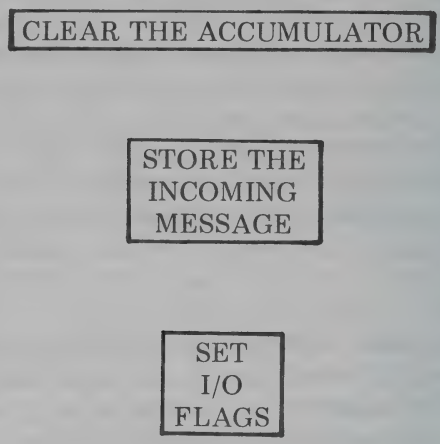
Microcomputer owners who plan to program their machines (rather than, say, purchase prepared programs from others) will probably find it wise to prepare special diagrams, called FLOW CHARTS, to guide them as they convert a program from the conceptual to the operational stage. Flow charts are a sort of shorthand method of illustrating the operation, or flow, of a computer pro-

gram. They also assist by graphically indicating the relationships between various parts of a complex program.

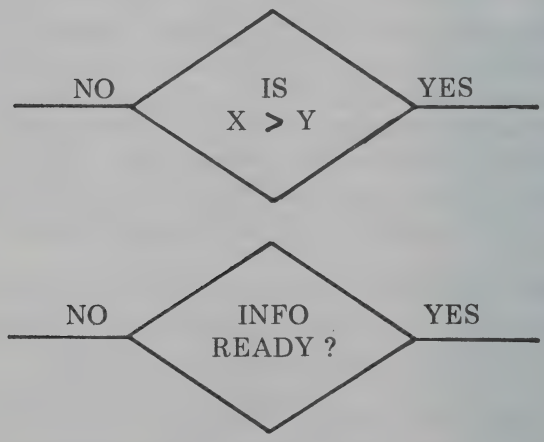
Flow charts are constructed by drawing lines between various types of symbols. The symbols denote certain classes of operations. Over the years, a variety of different symbols have been created for depicting the most commonly encountered types of computer operations. However, the beginner need not be concerned with becoming an expert in flow charting by learning a lot of flow chart symbols. Indeed, one can do quite well by just knowing a few of the most frequently used forms. The following simple symbols illustrate the principles and will allow the novice to flow chart programs quite adequately.

A circle may be used to specify the beginning or end of a series of related operations. Information may be displayed within the circle sign. The information may simply consist of the title of a routine, or it may indicate where a routine fits within a larger program. It might contain transfer information (indicating that the flow chart is continued elsewhere). Or, it might simply contain the statement START or STOP.

A square or rectangle may be used to denote a general or specific operation. The details of the operation may be described inside the box such as shown in the accompanying examples.

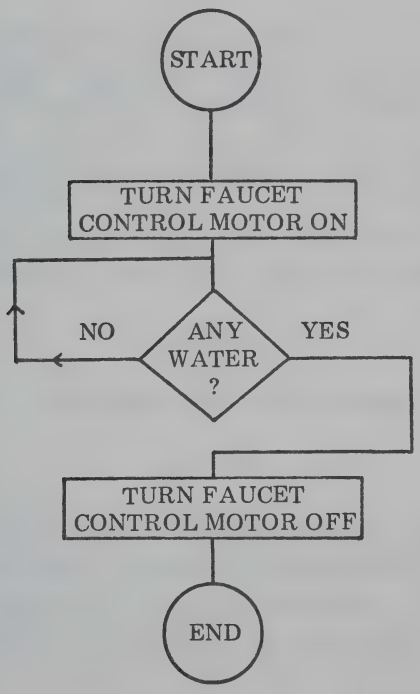


Only one other type of symbol is needed to be able to draw usable computer flow charts. The diamond shaped form illustrated next is used to indicate a decision point in a program. The two side points of the diamond form are used to indicate the path that operations will take when a decision has been made.



Lines with arrows are then used to connect the symbols to show the flow of operations as a program is executed. By popular convention, a flow chart is always read from top to bottom and left to right unless arrows on the lines indicate otherwise. Stated another way, this simply means that one only needs to place arrows on the lines that interconnect symbols if they are to indicate a flow to the left or up!

An example of a flow chart depicting the flow of operations for the algorithm mentioned earlier that could be used to have a computer control the operation of a water valve (turning it on in the example) is presented next. The concept of flow charting, while extremely simple, is also extremely effective. It assists in making sure that the computer programmer pays attention to detail as a program is prepared for the computer. As previously mentioned (and as will be made abundantly clear as this chapter progresses), there are a lot of details that the successful programmer must keep track of - especially when developing programs at the machine language level!



Just what kind of details does one have to know in order to do programming at the machine language level? How about having to memorize the eight bit binary values for each and every type of instruction a particular CPU can perform along with the different binary patterns for each different addressing mode that the instruction can utilize? Remember, that means knowing the meaning of several hundred different binary patterns for most types of CPU devices. (And knowing the patterns for one machine generally doesn't mean a thing if it becomes necessary to deal with a different CPU device!) How would that task be for a starter in getting down to details? It is enough to ruin most people's minds, thank you! Can you imagine trying to memorize a couple of hundred different patterns such as:

```
01 110 000
01 110 001
01 110 010
01 110 011
```

and maintain the relationship between each pattern and a specified meaning, such as "load the accumulator from CPU register B," "load the accumulator from CPU register C," and so forth? "Whew!"

That is just what a lot of other people said when computers were first developed. Who wants to do that? It didn't take long before someone said, "Look, let's make up a simple, easily remembered alphabetical abbreviation for each type of instruction, such as

LOD A,B

for "load the accumulator from CPU register B;"

LOD A,C

for doing the same from register C, and so forth. At least then people will have some kind of chance of being able to memorize synonyms for the various types of instructions." Other people thought the idea was pretty good (you better believe it!) and picked up on the concept. They called the alphabetical representations of

the binary patterns MNEMONICS for easily remembered (memory assisting) representations. The idea was extended to take care of different addressing modes so that, for instance, the representation of an IMMEDIATE mode of a load register directive might be depicted by the term LODI B,# ,where the LODI B mnemonic stood for the directive "load CPU register B using the immediate mode," and the "#" sign indicated that an actual (binary coded) number should be placed in the second byte of the two word instruction! The creation of mnemonics meant that, at least when creating a program, a human would be relieved from the drudgery of having to memorize hundreds of different binary patterns, and instead could concentrate on essentially the functional aspects of developing a routine. It is a lot easier for a person to think and write down a series of directives such as:

```
LODI A,#
 LOD B,A
 LOD C,A
```

in this case meaning "load CPU register A (the accumulator) with a number (stored in the next word in memory); then load register B from the contents of the accumulator; and finally do the same for register C." This is considerably easier than having to do something such as:

```
01 111 000
01 110 001
01 110 010
```

while trying to remember that those binary patterns meant just what has been described! For, the essence of the matter is that the mnemonic representations of the patterns, which are easily remembered because they assist the mind in recalling what directives they stand for, can also easily be arranged into an alphabetical table such as:

```
LODI A,#        01 111 000
   .
   .
LOD B,A         01 110 001
LOD C,A         01 110 010
   .
   .
```

so that, after one has arranged a sequence of directives using the mnemonic representations, one may rapidly look up the mnemonic symbols in the table and convert the mnemonics to the binary patterms that the machine must utilize. Then, when one actually desires to have the computer execute a program that has been created, the binary patterns representing the specific instructions can be loaded into successive locations in memory.

The mnemonics used to represent instructions are generally suggested by the CPU manufacturers. However, the choice of letters and symbols used to form mnemonics is really completely arbitrary from a practical point of view. If one does not like the mnemonics suggested by others, one may invent mnemonics to one's own likings. Generally, however, mnemonics for a given CPU become widely accepted. When this is the case, it is ordinarily best to learn the mnemonics in common use by others so as to facilitate communicating with other programmers. This is especially true if one intends to do a great deal of machine language level programming. This is because there are special kinds of programs (which will be mentioned further on in this chapter) which may be used to automatically perform the table look-up process to convert mnemonics to machine code patterns. These special programs are designed to recognize a commonly recommended or agreed upon set of mnemonics (usually the set recommended by the CPU manufacturer or the program developer).

O.K. What other kinds of details must a machine language programmer learn to deal with? (Some people might want to forget about becoming involved with this level of programming after reading the previous section! After all, to be truthful, learning all the mnemonics that represent the various instructions a microcomputer can perform, along with the variations to account for the different

ypes of addressing modes, can be a significant task in itself. That ask is just a small part of the overall effort that is involved in beoming proficient in this level of programming a microcomputer.) Another area in which people planning to do machine language programming should anticipate gaining some experience in is the ability o manipulate numbers in several numbering systems outside the ommonly used decimal, or base ten, system! Furthermore, one can lan on becoming proficient in converting values in one number ystem to a different number (base) system.

For instance, the reader is probably already aware of the need to ranslate between numbers expressed as values using the decimal base 10) manner of notation, and values expressed in the binary base 2) system. When dealing with computers, this conversion is necessary because computers manipulate numbers expressed as binary values. This is because the binary numbering system requires ust two digits (0 and 1) to represent any number. These two digits re readily symbolized in a machine by a switch being either in the off (0) or on (1) condition. On the other hand, humans are generally aught to manipulate numbers expressed in the decimal (base 10) numbering system because people are able to effectively manipulate much larger array of symbols (digits), and because manipulating numbers in that fashion is more efficient from several points of view. The person who wants to communicate with a computer at the nachine language level must learn to translate from the numbering system the human is well versed in to the one the machine is designed to manipulate effectively.

Strange as it may seem at first encounter, the ability for humans o deal with the binary numbering system utilized by the machine is generally facilitated by converting binary numbers to yet another numbering system (i.e., other than decimal numbers). This strange state of affairs is brought about by a human strength (or weakness, depending on your point of view) to be able to deal more readily with a larger number of symbols (digits), while having a weakness (strength?) for failing to be able to remember a large number of digits from a positional viewpoint. Thus, humans find it easier to remember a series of decimal digits such as 517 consisting of just three positions, even though each position may be occupied by one of ten different symbols, than to remember a series of nine binary

digits such as 100000101, even though each position can only assume one of two possible values!

If such be the case, then why not just deal in decimal numbers until such time as it is necessary to communicate with the computer, and then worry about converting to the binary system? Alas, the relationship between decimal numbers and binary representations is not as well ordered as some other numbering systems. The amount of disorder is such that it is often easier to deal in an intermediate numbering system that maintains a nicely structured relationship between itself and the binary system. Two such numbering systems are the OCTAL (base 8) and HEXADECIMAL (base 16). What is so nice about the relationships of these numbering conventions to the binary format? The fact that one digit position in the octal or hexadecimal system can always be mapped to correspond to a fixed number of digit positions in the binary format for all possible values.

Some further discussion should help clarify the relationships. The octal numbering system has eight digits (0, 1, 2, 3, 4, 5, 6, and 7). Any digit in the octal system can be uniquely represented by one of the eight possible patterns that can be contained in a series of three binary digits. Observe:

OCTAL DIGIT	BINARY PATTERN
0	0 0 0
1	0 0 1
2	0 1 0
3	0 1 1
4	1 0 0
5	1 0 1
6	1 1 0
7	1 1 1

If one maintains a fixed three to one ratio of digit positions between the binary and octal representations of a value, one can readily learn to quickly translate an octal value to its binary equivalent, or vice versa. For instance:

$$\overbrace{\underbrace{011}_{3}}^{3} \qquad \overbrace{\underbrace{100}_{4}}^{4} \qquad \overbrace{\underbrace{101}_{5}}^{5}$$

The human mind, given a little practice, can memorize the eight patterns (consisting of three binary digits) corresponding to each one of the eight octal digits. Once this has been accomplished, the job of translating from one numbering system to the other becomes a simple rote task of grouping and substituting appropriate digits. The system admirably serves the purpose of reducing the number of digit positions that must be remembered when performing intermediate processes involving the manipulation of numbers while developing machine language programs. The human can easily remember a value of 246 (or, at least, more easily remember it) while realizing at some future point that the code in the computer will actually be represented as the binary pattern 010 100 110 (grouping illustrated for emphasis). Conversely, a person can view a binary pattern such as 111010101 and readily group it mentally to the form 111 010 101 and remember it as the octal series 725.

The same type of principles apply to the hexadecimal to binary relationship except that the hexadecimal system uses 16 digits (generally defined as 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F), and a hexadecimal digit is mapped to four binary positions. The mapping relationship appears as shown here:

HEXADECIMAL DIGIT	BINARY PATTERN
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000

9	1 0 0 1
A	1 0 1 0
B	1 0 1 1
C	1 1 0 0
D	1 1 0 1
E	1 1 1 0
F	1 1 1 1

By maintaining a fixed four to one relationship between binary digit positions and hexadecimal digits, one may readily (but perhaps not quite as readily as when using the octal system) learn to convert binary values to hexadecimal values by grouping together binary digit positions and vice versa. For example:

$$\overbrace{1111}^{F} \qquad \overbrace{0001}^{1}$$

$$\underbrace{1111}_{F} \qquad \underbrace{0001}_{1}$$

Both the octal and hexadecimal systems are in common use among microcomputer programmers. The choice of which system to use is often one of personal taste, or the one to which a person is first introduced or trained to use. Note that the hexadecimal system allows eight binary digits to be represented by just two positions. (Remember, most microcomputer systems have eight bit wide registers.) But, there are more digits in the hexadecimal system and 16 patterns to remember. Also, some people have difficulty accepting the use of the letters A through F as numbers when using the hexadecimal method.

On the other hand, the octal system, while having less digits in the system (zero through seven), requires three digit positions to represent eight binary positions. Actually, three octal digits can represent nine binary positions. By convention, when dealing with eight bit wide computer systems, the binary contents of a computer register are considered to be grouped in the following fashion:

00 111 000

The most significant octal digit used to represent the three binary groups is restricted to the range 0 through 3 as though the implied third binary bit of the left-most group was always in the zero state. Some people find the octal system more convenient to work with as there are only eight patterns that have to be remembered in order to work effectively with the system, and there is no confusion with having letters represent numbers. However, as has been mentioned, the choice of which system to use is essentially one of personal taste. Of course, the computer owner will find literature that caters more heavily to one or the other system. The well rounded machine language programmer will naturally find it advantageous to be familiar with both conventions at the working level.

Not only will those planning on programming microprocessors at the machine language level find it necessary to become proficient at converting binary numbers to octal or hexadecimal series (and vice versa), they will also find it important to be able to convert from octal or hexadecimal to decimal, and to perform the reverse procedure. This type of procedure is not quite as neat as might be desired. However, it is not necessary to develop the conversion skill to the point where it can be done in one's head (as is often desirable when dealing with the conversions just discussed). In fact, for most people, such a practice would be an above average feat!

The reason the art of converting from decimal to octal or hexadecimal (and the reverse) is practiced is because it generally turns out to be easier to perform this operation, then expand the result to binary, than it is to convert directly from decimal to binary. (This point is debated by some people.)

How can one convert numbers expressed in the decimal numbering system to the octal system? There are several ways. Perhaps one of the easiest methods is described below.

Divide the decimal number by the number of digits in the octal numbering system - eight. Record the remainder of the division operation as the least significant digit of the octal number being derived. (Note that it is the remainder of the division operation that is being recorded as the least significant digit!) Use the quotient obtained from the division process just conducted as a new dividend.

Divide the new dividend by eight. The remainder from this division operation becomes the next significant digit of the octal number being derived. The new quotient is used as another new dividend. The process is continued until the quotient for a division operation becomes zero. The number obtained by arranging all the remainders from each division in increasing significant order (first remainder as the least significant digit, last remainder as the most significant digit) is the equivalent octal number value of the original decimal number. An example of the operation is illustrated below.

ORIGINAL NUMBER	1234 /	8	=	154	2
LAST QUOTIENT BECOMES NEW DIVIDEND	154 /	8	=	19	2.
LAST QUOTIENT BECOMES NEW DIVIDEND	19 /	8	=	2	3..
LAST QUOTIENT BECOMES NEW DIVIDEND	2 /	8	=	-	2...
OCTAL EQUIVALENT OF 1234 DECIMAL IS			=		2322

Reversing the conversion process to go from octal to decimal is straightforward. Simply multiply each octal digit by eight raised to its positional (weighted) power, and then add up the total product for all the octal digits. Observe:

2322 OCTAL =

...2	X	(8*0)	=	(2 * 1)	=	2
..2	X	(8*1)	=	(2 * 8)	=	16
.3	X	(8*2)	=	(3 * 64)	=	192
2	X	(8*3)	=	(2 * 512)	=	1024

DECIMAL EQUAVALENT OF 2322
OCTAL IS = 1234

A similar process to the one described above may be used to convert decimal numbers to hexadecimal values, or to perform the reverse operation. For the former conversion, one proceeds to divide the decimal number by the number of digits in the hexadecimal system (16). The remainder of each division becomes the next significant digit in the hexadecimal number being derived. The whole numbered result is repeatedly divided until its value becomes zero. The process is illustrated next.

ORIGINAL NUMBER	1234	/	16	=	77	2
LAST QUOTIENT BECOMES NEW DIVIDEND	77	/	16	=	4	D .
LAST QUOTIENT BECOMES NEW DIVIDEND	4	/	16	=	-	4 . .
HEXADECIMAL EQUIVALENT OF 1234 DECIMAL				=		4D2

Going in the other direction, one simply multiplies each hexadecimal digit by 16 raised to its positional (weighted) power, and then adds up the total of the products as shown next.

4D2 HEXADECIMAL =

. . 2	X	(16*0)	=	(2 * 1)	=	2
. D	X	(16*1)	=	(13* 16)	=	208
4	X	(16*2)	=	(4 *256)	=	1024
DECIMAL EQUIVALENT OF 4D2 HEXADECIMAL					=	1234

Now, readers who are students of mathematics might be asking themselves, "why not use the same kind of conversion technique to

go directly from decimal to binary, or vice versa?" Indeed, one could use the method, but look how drawn out it becomes:

ORIGINAL NUMBER	1234 / 2 =	617	0
QUOTIENT = DIVIDEND	617 / 2 =	308	1 .
QUOTIENT = DIVIDEND	308 / 2 =	154	0 . .
QUOTIENT = DIVIDEND	154 / 2 =	77	0 . . .
QUOTIENT = DIVIDEND	77 / 2 =	38	1 . . . .
QUOTIENT = DIVIDEND	38 / 2 =	19	0 . . . . .
QUOTIENT = DIVIDEND	19 / 2 =	9	1 . . . . .
QUOTIENT = DIVIDEND	9 / 2 =	4	1 . . . . .
QUOTIENT = DIVIDEND	4 / 2 =	2	0 . . . . .
QUOTIENT = DIVIDEND	2 / 2 =	1	0 . . . . .
QUOTIENT = DIVIDEND	1 / 2 =	-	1 . . . . .
THE BINARY EQUIVALENT OF 1234 DECIMAL IS:			10011010010

Readers will undoubtably agree that it takes considerably less work to obtain the octal or hexadecimal equivalent values of a decimal number, and then mentally expand those digits to binary representations. Note that the expansion of 2322 octal, or 4D2 hexadecimal, quickly yields the same binary value as that calculated above:

2	3	2	2
-10	011	010	010

4	D	2
- -10	1101	0010

One hardly needs to explain that a similar situation exists for going directly from binary to decimal notation. One may go through the tedious process of multiplying each binary digit by the value 2 raised to its positional power, and then sum the products to obtain the decimal value. However, by simply grouping the binary digits in-

to clumps of three or four, one can mentally form the octal or hexadecimal equivalent values, then proceed to perform far fewer steps to arrive at the decimal equivalent figure! (One must realize, of course, that when first introduced to the subject, the whole subject matter may seem somewhat complex. Given a little experience after carefully reviewing and studying the matter, most people would find the subject well within their grasp, though there is no guarantee that performing such manipulations will be within their concept of having fun. One of the purposes of this chapter is to give the reader some insights into the types of tasks one can reasonably expect to be involved with if one plans to program a microcomputer at the machine language level. If the types of tasks being described don't exactly get the reader all excited, one doesn't have to give up hope. Higher level programming (discussed in the next chapter) eliminates most of the kinds of details being presented now, at least as far as the computer user is concerned!)

What other types of details must a machine language programmer deal with on a regular basis? One very important detail that must be constantly paid attention to is where in memory instructions and data will be placed! Associated with this task is the ability to determine how much storage space will be required by data that is to be manipulated, and to judge how much space a particular type of program might require. Much of this type of judgement comes from experience, but certain fundamental concepts must be understood by anyone considering programming a computer.

First, a computer must be equipped with enough memory (in terms of the number of bytes or memory words available for use in the system) so that there will be room both for the sequence(s) of instructions that will direct the operations of the computer, as well as storage of the data that is to be processed. Sometimes determining the amount of memory required for the successful performance of a given task is not as straightforward as it might seem at first glance. Suppose, for example, that one has a computer system equipped with 2,048 (referred to as 2 K) bytes of memory. It is known, say that it is desired to input 1,024 measurements from an external device, and then to perform some calculation on the data obtained. Each measurement will require one byte of memory for storage. Sounds simple enough, right? Just set aside 1,024 bytes of memory

for the measurements and put the operating program (instructions that will direct the computer to obtain the data and perform the calculations) in the remaining 1,024 bytes of memory. But wait, what would happen if one of the calculations required that one keep track of the differences between successive measurements for each of the 1,024 readings? One would need an additional 1,024 bytes of memory to hold the additional data. If there were only 2 K of memory available in the system, there wouldn't be any room left for the instructions - or else the desired goal of storing the difference values would have to be eliminated.

But that is only part of the problem. How can one be sure that all the directives needed to obtain and store the data, and then perform the calculations, etc., will fit within a certain number of bytes of memory? The only way to tell for sure is to actually develop all the necessary routines and see if they fit. (With experience, one can learn to make fairly accurate estimates.) At first thought, one might suggest that one simply list all the instructions required to perform the desired operations, and then count them up to see if they number less than the number of memory locations available for program storage. (1,024 bytes in the hypothetical case mentioned above.) Ah but wait, from reading the previous chapter one knows that different instructions, or even the same types of instructions when using different addressing modes, may require one, two, or three bytes of memory in a typical microprocessor system! The machine language programmer must keep track of how many bytes of memory each individual instruction will utilize in order to obtain a total for a given routine or program.

To help them accomplish this task, many professional programmers rely on special forms, such as the one illustrated here, called PROGRAM DEVELOPMENT WORKSHEET. Note that each line of the sheet contains space to enter the machine codes related to each instruction. The form illustrated has room to associate up to three bytes for each instruction. It also provides columns for maintaining the actual addresses in memory. The column on the form titled MNEMONICS would be the column a programmer would first start working in when creating a program. The column to the right of the MNEMONICS column, termed COMMENTS, provides room for the programmer to make amplifying comments concerning details of the

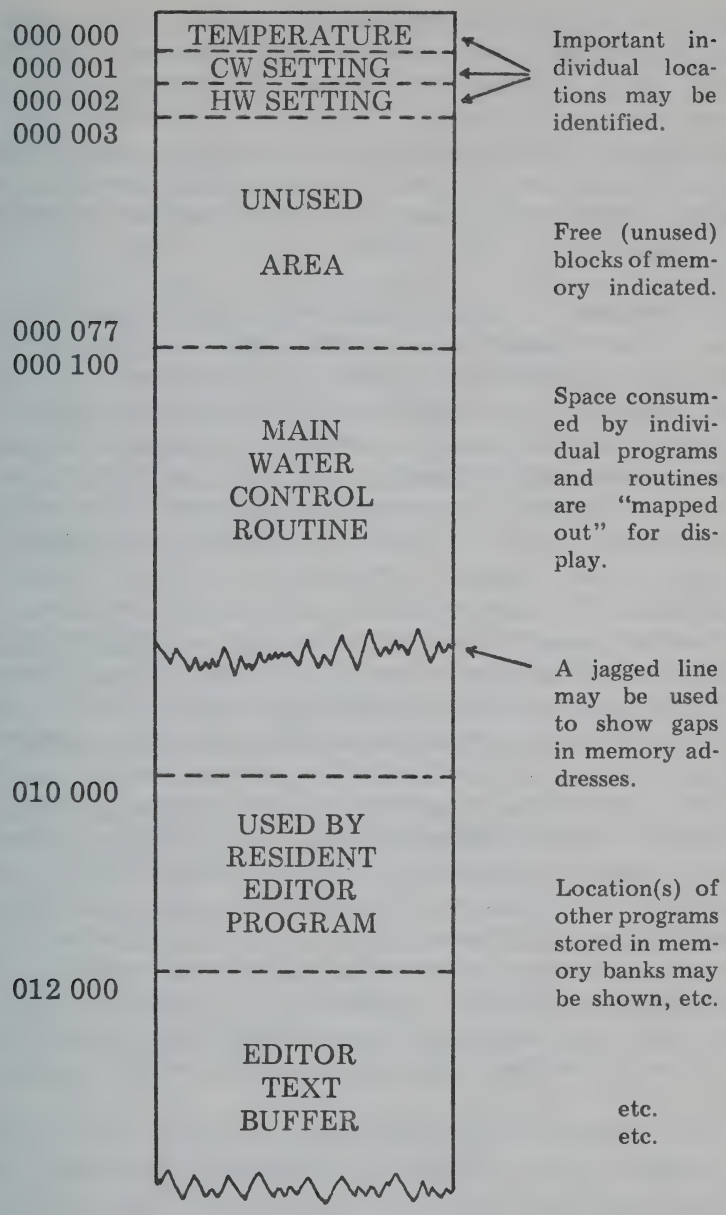
PG	LOC	MACHINE CODE			LABELS	MNEMONICS	COMMENTS
000	100	123	000	001	START,	LDI INDEX,000 001	Pntr to CW setting
000	103	017				LDX A	Fetch CW setting to Accum.
000	104	107				INR A	Increment CW value
000	105	333	001			OUT PORT1	Send new rate to controlier
000	107	117				STX A	Store current rate in memory

PROGRAM DEVELOPMENT WORKSHEET

program that might not be readily apparent from just following the series of instructions as they are listed. Finally, the column just to the left of the MNEMONICS column, designated LABELS, allows the programmer to assign NAMES to selected points in a program. The NAMES so assigned really refer to an actual memory address where a particular instruction will reside in memory. The use of a symbolic NAME instead of a series of digits representing an actual address is especially useful when initially developing a program. Why? Because it allows the programmer to refer to locations within a program without having to know where the location physically resides! It might be noted that by using just the columns titled LABELS, MNEMONICS, and COMMENTS on the example program development work sheet, one may logically create series of instructions - even series that might refer to other sequences - without having to specify the actual addresses in which the machine codes will reside, and without having to be concerned with the machine code patterns themselves during the program inventing process. This information can be supplied after the creative powers used to develop an algorithm have been (hopefully) successfully exercised. (More on this subject will be presented later in this chapter.)

To supplement the program development work sheet, it is often advisable for machine language programmers to draw up what are referred to as MEMORY MAPS. These memory maps are simply a way of graphically keeping track of what sections of memory are used (with, perhaps, supporting information concerning details of their use) by a program or a group of routines. Memory maps help programmers make sure that they do not start using sections of memory for one purpose when they are already being utilized for another purpose. This can be an important parameter to keep tabs on when a machine language program becomes large and/or complex. Memory maps, just as with flow charts, are extremely simple, yet effective devices. Prospective programmers should plan on making them a part of their programming work habits.

Details, details, so many details. That is what one may expect if one is going to be a successful machine language programmer. But, as one may be learning, others traveling the path (and, who know that the efforts can be most rewarding) have developed systems for easing



SAMPLE MEMORY MAP

the tasks of keeping the details in order. Flow charts, program development worksheets, and memory maps are some of the aids developed to assist in this area. A few others will be discussed shortly. But first, it is important to present another area (more details!) in which the potential machine language programming devotee can expect to have to acquire some working knowledge: codes for passing information in and out of the computer to external communication devices.

In the previous chapter the reader learned that most microprocessors communicate with external devices by passing information from the accumulator to a device (by executing an output instruction). Similarly, they are able to receive information from an external device into the accumulator (by executing an input instruction). Just what kind of information needs to be passed from the computer to an external device, or vice versa? In the case of controlling a piece of machinery, such as a motor to turn on a water faucet, the information can be essentially arbitrarily defined by the designer of the equipment. One can assign a particular bit position within the accumulator as a "motor on" indicator, and arrange things so that the status of that bit position will be relayed to an external switch. If the bit position in the accumulator is set to a logic '1,' and an output instruction is executed, the external switch becomes activated, or turns on. If the bit position in the accumulator is a logic '0,' and an output instruction is performed, the external switch goes off. Similarly, one can arrange the switch on a moisture sensing device so that it signals a logic '1' when the device indicates it is wet, and a logic '0' when it is dry. This switch can be connected to the computer via the input network such that when an input instruction is executed, the status of the switch will be transferred into a particular bit position within the computer. The entire arrangement can be essentially custom designed to suit the whims of the system owner, for it is assumed that the system is a customized application.

Since there are eight bits available for use in the accumulator of a microprocessor, the number of possible ways in which to connect up such a water faucet controller is rather large. If two people wanted to make water faucet controllers that could be controlled by the same program, they would have to make sure they agreed on which bit positions to assign to the switches (input and output), as well as to

conform to the same standard from the logic level point of view. That is, they would have to agree that a logic '1' indicated that a switch was on, and a logic '0' indicated that it was off.

Much of the communicating that takes place between a computer and external devices relates to the process of humans informing the computer what it (the human) wants the machine (computer system) to do, and the computer system telling the human what it (the machine) has done. This need, for practical considerations, has resulted in the creation of a variety of devices that can transform symbols that humans relate to (such as the letters, numbers, and symbols on a typewriter keyboard) into patterns of ones and zeros (binary numbers) that the computer is able to manipulate. Likewise, there are a variety of machines, electronic printers, video display units, etc., that can accept binary patterns (transmitted from the computer), and convert those patterns into symbols (letters, numbers, punctuation) that humans recognize readily.

As can be imagined, there are numerous ways in which one could conceive of designing systems to perform the above functions. If each person selected a different manner in which to accomplish the task, there would be little compatibility between different types of devices. Such a state of affairs is not conducive to mass production and the accompanying lowering of costs that people do enjoy! Far better, early computer pioneers said, if some form of systematic system for encoding information was developed so as to obtain standardization among the various producers of input and output devices. (Actually, the standardization process was developed to facilitate communications between various types of communication devices themselves, and was then sort of acquired by the computer pioneers.) In any event, there now exists several generally accepted coding schemes for passing information between various devices in a computing system. There are several such codes around because several codes had been widely accepted by several organizations before enough people had the good sense to get together and put a stop to the proliferation of different codes. They did this by agreeing to build equipment that used a common code. While there are several codes still in use, some of the codes are used by essentially commercial establishments that for one reason or another (most probably, economics constitute a major factor) have decided not to accept

the code that has gained national, indeed, international recognition and acceptance. This code is called the American Standard Code for Information Interchange, and is commonly abbreviated as the ASCII code. The ASCII code is the code with which most microcomputer machine language programmers should plan on becoming familiar.

Just what is the ASCII code? It is an agreed upon set of patterns, (128 such patterns in all) that can be represented by 7 binary bits. Each pattern is understood to stand for a particular character, such as a letter, number, punctuation mark, or special function or symbol. These patterns are used to encode information from transmitting devices, such as electronic keyboards. Similarly, these patterns will cause receiving devices, such as electronic typewriters, to display characters. A portion of the ASCII code is shown in the accompanying table. The table does not list all 128 of the agreed upon codes, but simply those most often encountered when communicating with a computer.

The reader might note that the ASCII code which requires just seven bits to be represented fits with room to spare in an eight bit wide microcomputer register or word in memory. How convenient. By convention, the seven bits are stored in the right most positions of a register. That leaves the left most bit (most significant bit) in an eight bit register available for other use. How nice. It turns out that computer programmers have found a good use in many cases for that "extra" bit.

For technical reasons, passing information from an external device to a computer is a weak link in a computer system. The extra long wires, mechanical and electrical interfaces, and the input device itself (such as a keyboard) greatly increase the chances that an electronic mistake may occur. What kind of electronic mistake? Well, the changing of a logic level from a '1' to a '0,' or vice versa. Such an unintentional change can raise havoc in a computer system. While such a change is not likely to occur in a properly operating computer, the chances that it may occur increase substantially when one must send the information over considerable distances (from one type of machine to another). When communicating with a computer,

ASCII CHARACTER SET

CHARACTERS SYMBOLIZED	BINARY CODE	CHARACTERS SYMBOLIZED	BINARY CODE
A	1000001	!	0100001
B	1000010	"	0100010
C	1000011	#	0100011
D	1000100	$	0100100
E	1000101	%	0100101
F	1000110	&	0100110
G	1000111	'	0100111
H	1001000	(	0101000
I	1001001	)	0101001
J	1001010	*	0101010
K	1001011	+	0101011
L	1001100	,	0101100
M	1001101	-	0101101
N	1001110	.	0101110
O	1001111	/	0101111
P	1010000	0	0110000
Q	1010001	1	0110001
R	1010010	2	0110010
S	1010011	3	0110011
T	1010100	4	0110100
U	1010101	5	0110101
V	1010110	6	0110110
W	1010111	7	0110111
X	1011000	8	0111000
Y	1011001	9	0111001
Z	1011010	:	0111010
[	1011011	;	0111011
\	1011100	<	0111100
]	1011101	=	0111101
↑	1011110	>	0111110
←	1011111	?	0111111
SPACE	0100000	@	1000000
CAR RET	0001101	RUBOUT	1111111
LINE FEED	0001010		

it is desirable for the computer to have some means of determining that there is a high probability that information it receives is accurate and has not been inadvertantly altered during transmission from an external device. The eighth bit of a microprocessor register may be used for this purpose when dealing with an external device that uses the seven bit ASCII code! How is this possible? Whenever the external device prepares a character for transmission by forming the appropriate ASCII code, it performs an additional step. It counts the number of binary positions that are set to the logic one state. It then manipulates the status of an eighth bit appended to the ASCII coded pattern so that the bit makes the total number of bits in the one state to be an odd or an even value (depending on the convention selected by the system designer). For instance, the ASCII code for the letter 'E' is:

1 000 101

If the system designer wanted to use an EVEN PARITY scheme, an eighth bit would be appended to the above code so that it appeared as:

11 000 101

Note that the pattern now has four bits set to the one state. The number of bits set to a one condition - four - is an even value. The eighth bit was appended to the original seven bit value in a logic '1' state so as to make the total number of bits set to a logic '1' condition in the byte be an even value. Note that if an even parity condition is desired, some characters would already have an even number of bits in the one condition. For instance, the code for the letter A is:

1 000 001

so that the eighth bit would simply be set to a logic '0' condition in order to maintain the number of bits in the logic '1' state at an even value:

01 000 001

One can just as readily establish a convention of ODD PARITY by reversing the role of the eighth bit. That is, the eighth bit may be manipulated so that the number of bits set to a logic one condition is an odd value. If such is the case, one may observe that the eighth bit would be set to a zero if the code for the letter E was transmitted:

1 000 101
01 000 101

while it would be set to a one if the code for the letter A was sent:

1 000 001
11 000 001

The choice of whether to utilize an even or odd parity is purely one for the system designer and equipment manufacturer.

What good does adding the eighth bit do when transmitting information? Suppose, due to a malfunction during the transmission process, one of the bit positions in the code pattern becomes altered from a one to a zero, or the reverse? Regardless of which bit position such an error condition occurs in, the total number of bits in the one state within the pattern will be changed from an even (or odd) value to the reverse state of odd (or even).

Remember in the previous chapter that a special CPU flag called the PARITY FLAG was discussed? This flag, it was mentioned, would be set or cleared after executing certain types of instructions as a function of whether or not the total number of bits in the logic one state within a register was odd or even. By establishing a consistent procedure within a computer system, one may use this CPU flag by testing the contents of a register each time it receives information from an external device to determine if an error condition has occurred during the transmission of an ASCII code pattern. For instance, if one is transmitting characters from a device that sets the eighth bit so as to maintain an even number of bits in the one state within a byte, and upon reception at the computer a test is made to see if the condition still exists, and it is found that it does not, one will know that an error has occurred during transmission. One may then take appropriate steps to have the pattern transmitted again, or

take other error-correcting measures. (It should be mentioned that parity testing does not absolutely guarantee that a pattern has been received correctly. It only increases the probability that an error can be detected. The parity method fails, for instance, if exactly two (or an even number of bits) changes state within a pattern during transmission!)

Why does the machine language programmer have to learn about the ASCII code? Because this code (or a similar type of code if one does not want to go along with the crowd) enables humans to communicate with a computer from a practical point of view! For instance, a person can type the letter L on an electronic keyboard. The keyboard can convert the letter L to the ASCII binary pattern 1 0 0 1 1 0 0. The binary pattern can be transmitted to the computer (perhaps after adding a parity bit) where it will arrive in the accumulator. Now - and here is the important part - the computer can be programmed by the machine language programmer so that it can recognize that binary pattern to mean that it is to perform a certain task. Perhaps it would mean "list the contents of a section of memory." That section of memory might contain data that has been collected from an instrument making measurements. In any event, the reader can see that a logical system now exists for translating from symbols that humans understand (letters of the alphabet, numbers, punctuation) to symbols that the computer understands (binary patterns consisting only of ones and zeros).

Or, the computer can be programmed to store the ASCII codes it receives directly into memory (in order to store messages). It can then later recall those patterns and transmit them to an output device that would convert the ASCII patterns back into human recognizable symbols! All that has been said here about receiving characters from an input device also holds true going from the computer to an output device. A computer may be programmed to form the appropriate ASCII codes (and set the eighth bit to conform to a selected parity convention if desired), and then transmit the binary code to an output device, such as a video display unit, that will convert the ASCII codes to the symbols for which they stand.

The individual intending to direct the operation of a microcomputer at the machine language level should thus plan on becoming

familiar with the ASCII code, for this code is the most commonly used means of passing information from computers to people, and from people to computers.

Are there still more details that a machine language programmer can expect to be concerned with when developing programs? Yes. One frequently related to inputting and outputting information to/from a computer relates to knowing how much time it takes to perform a specific type of instruction or series of instructions. While a computer operates at very fast speeds, and can typically execute individual instructions in just a few millionths of a second, it does take a finite amount of time. Quite often it is necessary to calculate just how much time it will take the computer to perform specific operations. Thus, the programmer dealing with problems at the machine language level can expect to have to learn the lengths of time that a particular computer takes to perform each type of instruction. This can be a function of the addressing mode that an instruction is utilizing, and can even be affected by the type of memory devices used in a system and other considerations. What it comes down to is that the machine language level programmer can expect to have to learn a lot of facts about the true operating speed of the type of computer being programmed.

Why is this necessary? Suppose a computer is to take readings from an instrument, store the readings in memory, and perform calculations on the data obtained, such as maintaining a running average of readings over a given time span. If, for example, the instrument has a new reading ready for the computer every millisecond (one thousandth of a second), the computer must be capable of processing individual measurements and performing the necessary calculations within that time period. That means the program directing the operation of the computer must be arranged in such a manner that execution of all the necessary directives will not take longer than the allotted time period. To accomplish this, the programmer must determine how long it will take the computer to perform all the related operations. This information must be calculated as a function of the specific instructions utilized, etc. These types of calculations can indeed, become quite complex. The microcomputer programmer that intends to develop input/output related routines, or programs that control machinery, or that receive information from instruments,

should plan on becoming acquainted with computer timing information, and be prepared to handle the details associated with creating programs that require complex evaluation in that regard.

Getting tired of reading about all the details one can expect to be dealing with if one is going to program microcomputers at the most fundamental level? Well, at this point, the reader has been informed of enough to get started in the field. In actuality, one can spend a lifetime learning about aspects of machine language programming. In fact, for many people, that is the fun of the activity. However, beyond a certain point of learning the tools of the trade, such as those that have been mentioned here, the challenge of machine language programming becomes directly related to the problem of being able to define algorithms, and successfully implementing those algorithms with the instruction set available in a particular computer.

Since machine language programming seems to require acquiring quite a body of detailed knowledge, which, frankly, amounts to quite a bit of work, is it worth it?

First, if not already apparent, all computer operations must be derived from this level of programming! High level language operation, such as will be discussed in the next chapter, is obtained by someone first developing every part of the capabilities of the high level language using machine language. The reader who decides to stick essentially with that level of programming should at least appreciate the fact that somebody (more likely, a group of bodies!) had to go through the painstaking task of building up the high level language capabilities using machine language techniques. (The task is by no means trivial for a good high level language. It can take several many years and more of hard work to develop original versions of higher level languages.) So, without some people making the effort to develop higher level languages, the use of the computer would be pretty well restricted to just that group of people that had the capability and the fortitude to learn all the kinds of details being discussed in this chapter. That would indeed be a pity. So, yes, from the point of view of opening up the use of computers to the general population, the fact that at least some people learn the details of machine language programming is certainly beneficial.

But, really now, the reader probably wants to know whether there is very much prospect that going to the trouble of learning machine language programming will be beneficial to the individual who plans on having his or her own computer system. If not necessary, is it even desirable?

Ahah - the answer to that, not suprisingly, has a lot to do with just what services the prospective computer owner desires to have the machine perform!

If one plans on simply using a computer system as a simple calculator-like device, to solve problems that can be readily stated in the form of general mathematical statements, formulas, etc., using the power of the computer to quickly perform iterations, permutations, and so forth, then one will probably do quite will with a system that simply executes a high level language that someone else has prepared for the machine.

If one just needs to sort data, or search and extract information from a data base, or keep track of accounts, or perform other routine tasks of a nature that many other people perform on a computer, chances are one can get by without knowing anything about machine language programming. This is because commercially prepared programs are generally available, in the form of high level language, or special "operating packages," that will perform such services.

If one plans to buy every part of a computer system from a reliable supplier, and that supplier also provides all of the operating software (programs) for all of the types of capabilities one desires to have on the computer system (or at least, other suppliers can provide such programs), then one does not have to worry about doing machine language programming.

But, if a person has plans to do anything unusual or unique. If a person intends to maintain their system. If a computer owner plans to use the system as an equipment controller. If one wants to mix and match pieces of equipment within a system (such as obtaining the computer from one supplier, input and output devices from another supplier). If one wants to custom tailor a computer system to suit the specific needs of an individual owner. Then, chances are

good that person will find a working knowledge of machine language programming most helpful, if not mandatory!

What it comes down to is that there are two major classes of computer owners (users). Those that strictly limit their use to utilizing a computer system as a "ready to operate" tool to perform specific functions or tasks without having to be concerned with the technical aspects of how the machine or the machine language program that directs the basic operation of the system operates. This level of computer usage is analogous to the manner in which the general population considers cars. They are vehicles that a person drives around to accomplish specific tasks. Put in gas and drive it - put in a high level program (as discussed in the next chapter) and operate it! This is the level at which most computer system owners probably prefer to operate. (Problem was, up until recently, most private computer owners could not readily obtain (afford) such capabilities. Times, however, are rapidly changing.) The other category of user may be likened to the car buff who likes to keep the engine tuned up. The user that may be interested in using the vehicle for special purposes (drag racing, show, recreational vans). The user who wants to know how the engine operates. Who possibly has ideas on how to improve engines, or transmissions, and who wants to work at the nuts and bolts level. This is the level at which machine language level programmers might be considered.

The benefits of being able to perform machine language programming are similar to those obtained by the person who can work at the nuts and bolts level of an automobile. Such a person can certainly enjoy the benefits and joy of just driving a car, and using it to get from one place to another, However, if something goes wrong, the person able to work at the nuts and bolts level may have some advantages. Such as, for instance, not having to pay someone who does have that kind of knowledge seemingly outrageous fees to tell them what is wrong and fix it. Or, being able to fine tune an engine so as to obtain maximum performance for a given task. Or, being able to install a superior clutch system from one make or model into a unit that has a superior engine so as to obtain the best of two products. The machine language programmer is likely to find similar types of benefits accrueing on account of their additional knowledge and capabilities.

If the reader has not been overwhelmed by the prospect of learning all the various types of detailed information mentioned in this chapter, perhaps the reader will find programming a computer at this most fundamental level an enjoyable pastime. If the reader has experience at a practical level in digital electronics, and likes to putter around with, for instance, designing electronic logic circuits using IC (integrated circuit) devices, it is likely that the reader has an inclination towards the thinking processes that are beneficial at the machine language level of programming. The same might be said for those that lean towards an interest in mathematics (particularly if they appreciate Boolean algebra), and especially those that enjoy solving complex mathematical problems.

On the other hand, if the reader's main interests are oriented towards the liberal arts area, or the reader has trouble adding up three sets of two digit numbers accurately, the reader might be better off to plan on using applications programs that others have prepared in the reader's area of interest. Or, if they are curious, or want to develop original programs in their area of interest, to try high level programming (discussed in the following chapter) before they plunge into a big machine language programming project!

Before closing this chapter, it might be interesting for some readers to observe the development of a small illustrative program at the machine language level. This demonstration may help crystalize some of the points that have been discussed concerning the detailed aspects of programming at this level. Then, too, a practically oriented example may show some readers the type of fun that may be had when developing a program to solve a challenging problem.

The hypothetical problem that will be used for demonstration purposes is an expansion of the concept of controlling a water spigot using a microcomputer. Instead of just turning a water faucet on, the example will discuss controlling two faucets so as to fill a tank with water at a specific temperature.

The first thing a programmer must do when preparing to program a microcomputer is to decide just exactly what needs to be done. The best way to approach this aspect is to write down the details of the problem. The hypothetical problem might be described by a

programmer as follows.

The computer is to control two water valves, one for hot water, one for cold water. The actual controls are arranged such that electronic signals from the computer will cause them to be set to a position corresponding to a numerical value ranging from zero (off) to the maximum setting that may be represented by eight binary digits (255 decimal) that would represent maximum water flow.

The computer is to constantly adjust the flow rate of the hot and cold water so as to fill a tank with water at a specified temperature. (The desired temperature is to be stored in a location in the computer's memory prior to the start of operations.) An electronic temperature measuring instrument will be connected to the computer to enable the computer to constantly check the actual temperature of the water in the tank. This instrument will present the readings as binary data on seven electronic wires that will be transferred to the accumulator whenever a specific input instruction is executed. The data will arrive in the seven right-most bit positions in the accumulator.

Whenever a temperature reading is obtained by the computer, the eighth bit position (most significant bit) will be set to a logic '1' condition if a moisture detector at the top of the tank being filled is wet. If dry, the bit will be a logic '0.' When the computer determines that the tank is full, it is to shut off all water flow to the tank (by sending the binary value zero to the valve controllers).

The accompanying pictorial illustrates the essential physical components of the system that is to be controlled.

Once a programmer has clearly defined in general terms what needs to be accomplished, it is good practice to break the project down to a series of small steps showing just what needs to be done in order to accomplish the desired objective. This practice enables one to carefully review the "logic" of the operations to make sure they are feasible, and that they relate properly to other steps. The task of controlling the water faucets might be more clearly defined (than that indicated in the essay that described the general operation) by the following series of statements.

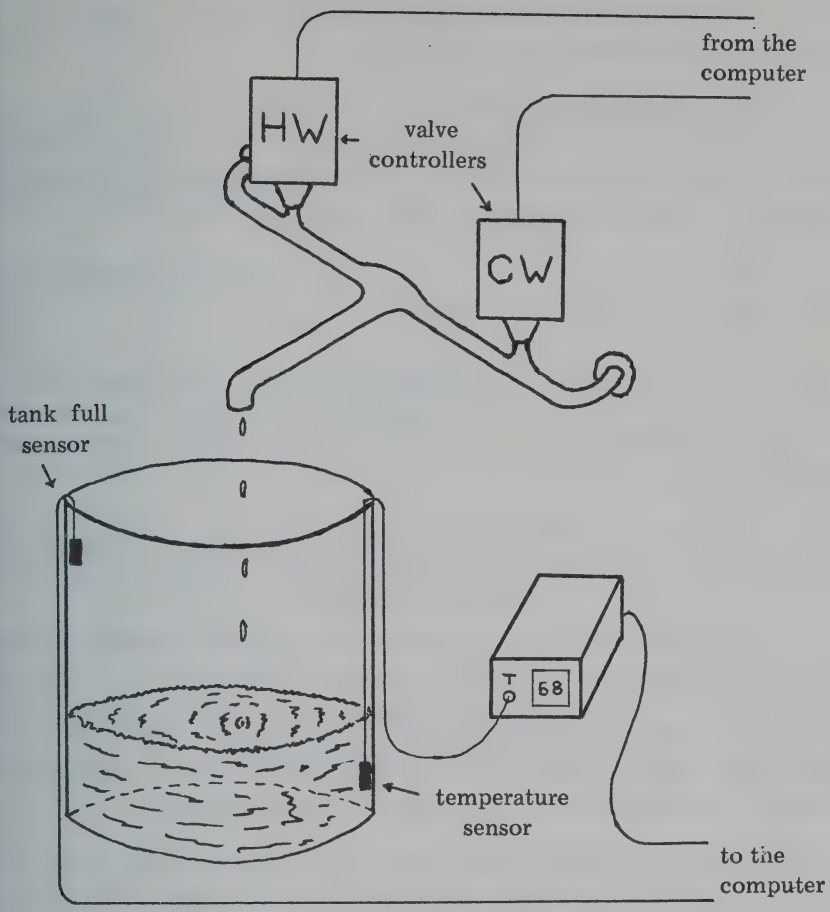


DIAGRAM OF A CONSTANT TEMPERATURE TANK FILLING SYSTEM ARRANGED FOR CONTROL BY A MICROCOMPUTER

1.) Start (or continue) operations by increasing the value of the cold water (CW) controller setting (stored in a data storage area in memory), and output the new value to the valve controller.

2.) Input data from the electronic thermometer and tank full indicator.

3.) Test the data obtained to see if the tank is full. If so, go directly to perform step number seven.

4.) Compare the data obtained for the temperature of the water in the tank against the data stored in memory as the desired temperature. If the temperature reading obtained matches the temperature desired, jump back and proceed from step number two.

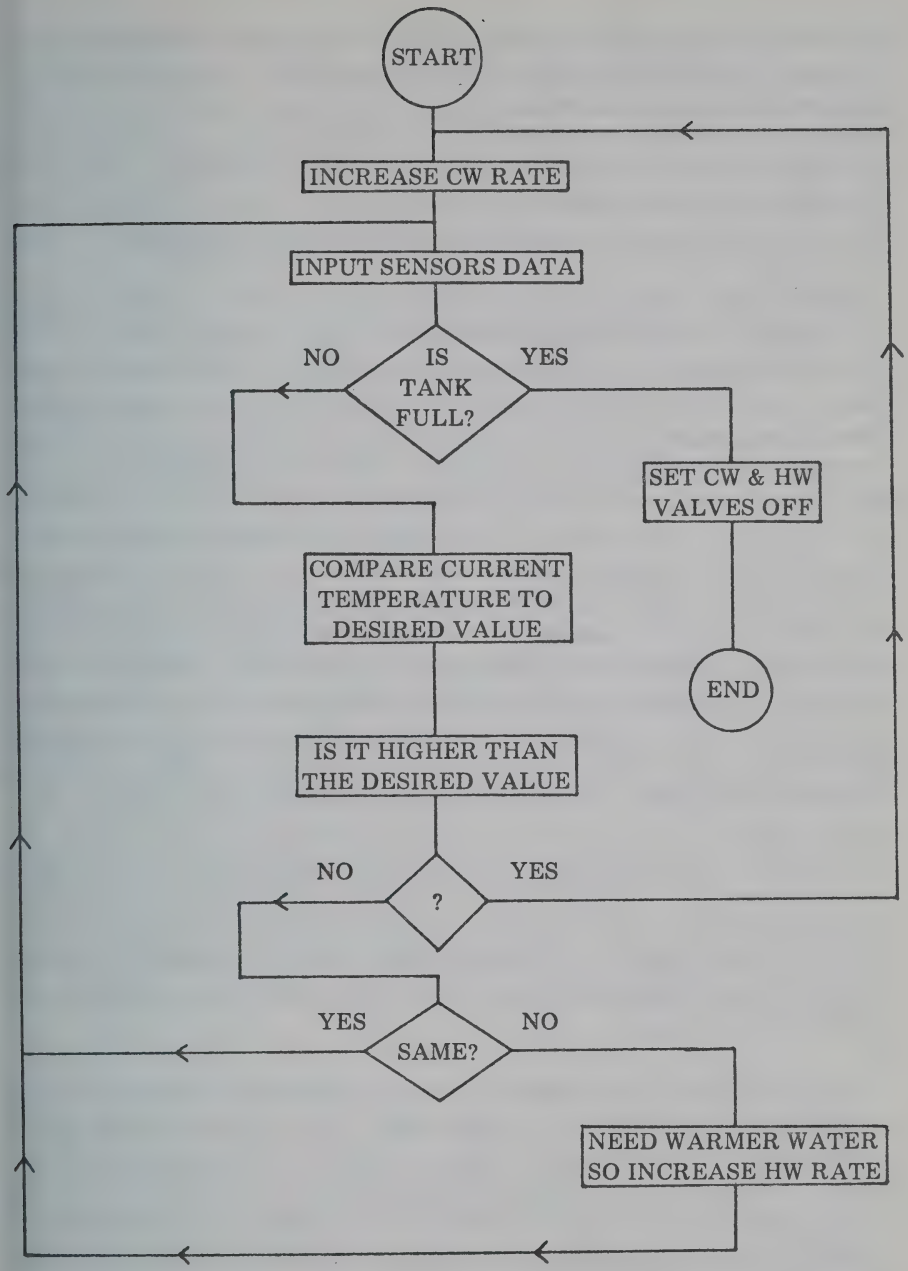
5.) If the reading obtained is higher than the desired setting, jump back and proceed from step number one.

6.) If the reading is lower than the desired temperature, increase the value of the hot water (HW) controller setting (stored in a data storage area in memory), and output the new value to the valve controller.

7.) Set both the CW and HW controller settings to a value of zero (no flow) and output the information to the valve controllers.

At this point in the development of a machine language program, it is a good idea to present the information that has been thought out in the form of a flow chart. A flow chart serves to illuminate the relationships between different parts of a program, such as the action paths that may be taken at decision points. The flow chart accompanying this discussion illustrates this beneficial aspect.

Once the programmer has decided precisely what needs to be done in a program, and outlined the steps (verbally and supported by a flow chart as has been illustrated), it is time to begin selecting the actual individual machine language instructions that will be used to direct the computer's operation. These instructions need to be stored in the computer's memory in proper sequence(s). This is because once the computer is placed in operation, the program counter in the CPU will automatically increment to point to the next location (address) in memory unless a special category of program counter altering instructions (such as a jump or call directive) is encountered. However, the actual instructions that direct the computer's operation are not the only pieces of information that typically must be placed in memory. Many times, as in the example being discussed, it will be necessary for DATA (such as the values re-



TANK FILLING PROGRAM
FLOW CHART

presenting the settings of the electronic valve controllers, the temperature readings obtained from the electronic thermometer, and the desired temperature at which the water in the tank is to be maintained) to be stored in locations in memory. This data that is to be stored in memory must be kept separated (at different addresses in memory) from the locations that will be used to hold the sequences of instructions used to direct the operation of the computer!

To facilitate keeping track of what areas in memory will be used for various purposes, programmers prepare memory maps (as has been mentioned previously) that depict blocks of addresses showing the general use of various sections of available memory within a computer system. While the hypothetical problem being discussed only requires three locations in memory for holding data, it is possible that more locations in memory might be required for such purposes if the program was expanded, etc. Thus, for illustrative purposes, it was decided to reserve a block of 64 decimal or 100 octal (addresses 000 to 077 in memory using octal address notation!) locations for storing program related data. The actual operating instructions for the program would thus begin at address 100 (octal, which would be address 64 decimal, 00 100 000 in binary as it would appear in an eight bit binary register), and extend as far as required in memory beyond that address. This information could be conveyed by drawing a memory map such as that shown on page 23 in this chapter.

Once the groundwork of verbally stating the problem, analyzing and proposing a solution using words, flow charting the proposed solution, and deciding where in memory various parts (instruction sequences and data) should reside, one is ready to create the actual series of instructions that will direct the computer to perform the desired operations.

The first step in this highly creative process is to select the individual instructions that the programmer feels will do the job and to arrange them in the desirable logical sequence. This is accomplished using the mnemonics that represent the actual instructions that the CPU being utilized is able to execute. This selection and arrangement process may be initially accomplished by writing down the mnemonics for the directives one desires to have the machine perform.

Often while working at this mnemonic level it is necessary to refer to other sections of a program. That is, to direct the computer to JUMP to various instructional sequences (or CALL subroutines) that start at certain addresses in memory. Since, during the development stage, it is often not possible to know exactly where portions of a program will reside in memory to an exact address, programmers have developed the practice of referring to addresses in memory by making up SYMBOLIC NAMES or LABELS. These symbolic names or so-called "labels" may be fashioned so that they have a tendency to aid the programmer in remembering what various sections of a program are designed to do. (Just as mnemonics serve to remind programmers what purposes the various instructions serve.) Labels (symbolic names used to indicate an address) will eventually be converted to binary patterns representing actual machine memory addresses - just as mnemonics will be translated to their binary representations utilized by the computer.

The following series of instructions illustrates one manner in which a programmer might decide to program a typical microcomputer in order to direct it to perform the operations necessary to control the hypothetical tank filling problem proposed. Hypothetical mnemonics are used to represent various kinds of instructions typically provided in microprocessors (as was discussed in the previous chapter). A COMMENTS column is provided to explain the purpose of each instruction selected. Additionally, the example groups sequences of instructions together as they relate to the numbered paragraphs outlining the program's operation.

①	START:	LDI INDEX,000 001	Load Index Register with the address in memory that is to store the cold water (CW) valve setting.
		LDX A	Fetch the value in the memory location that is pointed to by the address held in the Index Register into the Accumulator Register.
		INR A	Increment that value (which is the (CW) valve setting) so as to increase the flow rate.

①		OUT PORT1	Send the new (CW) rate setting to the valve controller.
		STX A	Store the new flow rate value (CW) back in memory at the address pointed to by the contents of the Index Register.
②	TEMPIN,	INP PORT3	Accept a new data reading from the thermometer and "tank full" indicator.
③		JTS FULL	Test the status of the SIGN FLAG (most significant bit in a register) assuming that the INPUT operation just performed affects the CPU flags. If the MSB (SIGN FLAG) is now set to a logic '1' it means the tank is filled. In that case the program should JUMP to follow the directives beginning with the address labeled "FULL."
④		LDI INDEX, 000 000	If the SIGN FLAG was a logic '0,' tank is not full. So, continue by pointing to the memory address of the location in memory where the desired temperature is stored.
		CMP INDEX	Compare present temperature reading in the Accumulator against desired temperature stored at address pointed to by the Index Register.
		JOZ TEMPIN	Jump On Zero - that is: If the actual temperature in the tank equals the desired temperature setting then no change in flows required - just go back and monitor the tank status!

5		JFS START	Jump if False Sign Flag, which in this case would mean that if the actual tank temperature was warmer than the desired temperature, then JUMP back to the address in the program labeled START so as to increase the flow rate of the cold water.
6		LDI INDEX,000 002	If temperature not just right or too hot then it must be too cold. So, set the value in the Index Register to the address of the location in memory where the hot water (HW) valve setting is stored.
		LDX A	Fetch the current value of the (HW) setting into the Accumulator.
		INR A	Increment the (HW) setting so as to increase the flow rate of the hot water.
		OUT PORT2	Send the new (HW) rate setting to the valve controller.
		STX A	Store the new flow rate (HW) back in memory.
		JMP TEMPIN	Jump back and get a new reading on the tank's status.
7	FULL,	LDI INDEX,000 001	Set the Index Register to contain the address of the (CW) valve controller storage register in memory.
		CLR A	Special instruction generally provided in microprocessors to clear the Accumulator to a value of zero.
		STX A	Store the value zero from the Accumulator to the (CW) storage location in memory.

⑦	OUT PORT1	Also output the value zero to the (CW) valve controller to shut off the flow of cold water.
	INR INDEX	Increment the address in the Index Register so that it points to the (HW) storage byte in memory.
	STX A	Store the value zero in the (HW) storage location.
	OUT PORT2	Also shut off the flow of hot water.
	STP	Special instruction to STOP operation of the computer as task of filling the tank has been completed!

Note in the above listing that symbolic names (labels) are used to identify key addresses in the program that other instructions (the JUMP category) need to reference.

In actual practice, a sequence of instructions making up a program could be placed directly onto a program worksheet of the kind mentioned earlier in the chapter. The next several pages illustrate how the above list of instructions might appear when initially placed on such program development worksheets.

By now the reader is well aware of the fact that a CPU operates by decoding binary patterns of information which it recognizes as individual machine language instructions. The information shown on the worksheets illustrated on the next several pages does not yet contain any such patterns. However, there are columns provided on the worksheets for such information The reader may note that once the proper sequence of instructions has been determined for a program it will be a relatively easy matter, using the program worksheets, to convert the mnemonic information on those sheets to the numerical information required by a microcomputer.

PG	LOC	MACHINE CODE			LABELS	MNEMONICS	COMMENTS
000	000				TEMP,		Storage loc of desired temp
000	001				CWCONT,		Storage loc for CW setting
000	002				HWCONT,		Storage loc for HW setting
000	100				START,	LDI INDEX,000 001	Pntr to CW setting
						LDX A	Fetch CW setting to Accum
						INR A	Increment CW value
						OUT PORT1	Send new rate to controller
						STX A	Store current rate in memory
					TEMPIN,	INP PORT3	Accept new sensor data
						JTS FULL	If tank full, shut off water!
						LDI INDEX,000 000	Pntr to desired temperature
						CMP INDEX	Compare data to desired

EXAMPLE PROGRAM AS IT WOULD INITIALLY
APPEAR ON A PROGRAM DEVELOPMENT WORKSHEET

PG	LOC	MACHINE CODE			LABELS	MNEMONICS	COMMENTS
						JOZ TEMPIN	If just right, form steady loop
						JFS START	If too hot, increase CW flow
						LDI INDEX,000 002	If too cold, increase HW flow
						LDX A	By fetching current HW value
						INR A	To Accum & increasing it
						OUT PORT2	Send new value to valve
						STX A	And save new value in memory
						JMP TEMPIN	Continue monitoring ops
					FULL,	LDI INDEX,000 001	Set pntr to CW valve setting
						CLR A	"Clear" Accum to all zeros
						STX A	Store zero in CW storage loc
						OUT PORT1	Send 0 to shut off CW flow
						INR INDEX	Increment pointer value
						STX A	Store zero in HW storage loc
						OUT PORT2	Send 0 to shut off HW flow
						STP	Halt computer operation

EXAMPLE PROGRAM AS IT WOULD INITIALLY
APPEAR ON A PROGRAM DEVELOPMENT WORKSHEET

All that needs to be done now is to look up the binary pattern associated with each instruction mnemonic and show that pattern residing in the appropriate location in memory on the worksheet. Since some types of instructions refer to other instructions in memory (or data locations), it is also necessary to convert labels (symbolic names) to actual binary address values. Is it necessary to use binary notation on the worksheets? Having to write down each binary pattern on the worksheet (eight digits for each word in memory utilized) could be quite messy, subject to error, and downright tiring on the fingers. An easier way to proceed for purposes of completing the worksheet is to use the octal or hexadecimal equivalent values for addresses and instruction codes! This saves having to write down a lot of binary ones and zeros. And, the reader now knows that expanding octal or hexadecimal digits to groups of three or four binary digits is a task that can be done mentally with just a little practice when it comes time to actually load the program into the memory of a computer.

The process of converting a program from a mnemonic listing to the actual machine code patterns is called ASSEMBLING a program!

Assembly of a program may be viewed as a two part procedure. The first part consist of allocating the proper number of bytes for each kind of instruction and as a function of this information determining the address of each instruction in a program that has been assigned a label. (While allocating the proper number of bytes for each kind of instruction one can also perform the actual task of determining the machine code pattern for instructions, data, and other values that are not dependent on addressing information. Obtaining the machine code pattern for an instruction is a simple matter of looking up the information in a table provided by a CPU manufacturer or the originator of the mnemonics being utilized by the programmer. The code patterns for data or operand portions of instructions are numerical values representing the data or operands - such as the immediate portion of a "load register immediate" directive.)

The second part of the typical assembly operation consist of using the addressing information that is available after the first part (described in the preceeding paragraph) has been completed. This addressing information is used to complete the code for multi-byte

PG	LOC	MACHINE CODE			LABELS	MNEMONICS	COMMENTS
000	000	104			TEMP,		Storage loc of desired temp
000	001	000			CWCONT,		Storage loc for CW setting
000	002	000			HWCONT,		Storage loc for HW setting
000	100	123	000	001	START,	LDI INDEX,000 001	Pntr to CW setting
000	103	017				LDX A	Fetch CW setting to Accum.
000	104	107				INR A	Increment CW value
000	105	333	001			OUT PORT1	Send new rate to controller
000	107	117				STX A	Store current rate in memory
000	110	366	003		TEMPIN,	INP PORT3	Accept new sensor data
000	112	360	——	——		JTS FULL	If tank full, shut off water!
000	115	123	000	000		LDI INDEX,000 000	Pntr to desired temperature
000	120	223				CMP INDEX	Compare data to desired

PARTIALLY ASSEMBLED EXAMPLE PROGRAM
(NOTE MISSING ADDRESSES IN "JUMP" INSTRUCTIONS)

PG	LOC	MACHINE CODE			LABELS	MNEMONICS	COMMENTS
000	121	340	———	———		JOZ TEMPIN	If just right, form steady loop
000	124	320	———	———		JFS START	If too hot, increase CW flow
000	127	123	000	002		LDI INDEX,000 002	If too cold, increase HW flow
000	132	017				LDX A	By fetching current HW value
000	133	107				INR A	To Accum & increasing it
000	134	333	002			OUT PORT2	Send new value to valve
000	136	117				STX A	And save new value in memory
000	137	304	———	———		JMP TEMPIN	Continue monitoring ops
000	142	123	000	001	FULL,	LDI INDEX,000 001	Set pntr to CW valve setting
000	145	377				CLR A	"Clear" Accum to all zeros
000	146	117				STX A	Store zero in CW storage loc
000	147	333	001			OUT PORT1	Send 0 to shut off CW flow
000	151	127				INR INDEX	Increment pointer value
000	152	117				STX A	Store zero in HW storage loc
000	153	333	002			OUT PORT2	Send 0 to shut off HW flow
000	155	000				STP	Halt computer operation

PARTIALLY ASSEMBLED EXAMPLE PROGRAM
(NOTE MISSING ADDRESSES IN "JUMP" INSTRUCTIONS)

classes of instructions that refer to memory addresses (such as JUMP and CALL directives).

Hypothetical octal equivalents of the binary patterns representing the machine codes for the various types of instructions will be used in the sample program. Memory addresses will also be expressed as octal values for consistency.

Doing the first part of the assembly process on the sample program would result in the worksheets appearing as just illustrated.

Note that boxes marked with a line on the worksheets indicate bytes of an instruction that will eventually contain addressing information. As work proceeded on the worksheets (determining the number of bytes required by each instruction) it became possible to determine the actual addresses of instructions designated (and referenced) by labels. With that information now known it is possible to go on to complete the assembly of the example program by filling in the memory locations (indicated by the placement of a line in the appropriate spaces on the worksheets just presented) with actual addresses. The resulting completely assembled program ready to be loaded into a microcomputer's memory is shown on the next several pages!

At this point, one would be in a position to load the memory banks of a computer (hypothetical in this case) that would be capable of decoding the patterns and performing the indicated operations that they represented. Remember, the octal values shown on the worksheets would be expanded to represent eight binary bits during the memory loading process. (Note: How the patterns could actually be loaded into memory is a matter that will be discussed later in this book.) Once in memory, the program counter of the CPU would be set to contain the address 000 100 (octal) and the computer placed in the program execution or "run" mode. Execution of the program would then be automatic until the tank was filled. The example demonstrates how a microcomputer may be "taught" how to fill a tank with water at a specified temperature!

PG	LOC	MACHINE CODE			LABELS	MNEMONICS	COMMENTS
000	000	104			TEMP,		Storage loc of desired temp
000	001	000			CWCONT,		Storage loc for CW setting
000	002	000			HWCONT,		Storage loc for HW setting
000	100	123	000	001	START,	LDI INDEX,000 001	Pntr to CW setting
000	103	017				LDX A	Fetch CW setting to Accum.
000	104	107				INR A	Increment CW value
000	105	333	001			OUT PORT1	Send new rate to controller
000	107	117				STX A	Store current rate in memory
000	110	366	003		TEMPIN,	INP PORT3	Accept new sensor data
000	112	360	000	142		JTS FULL	If tank full, shut off water!
000	115	123	000	000		LDI INDEX,000 000	Pntr to desired temperature
000	120	223				CMP INDEX	Compare data to desired

COMPLETELY ASSEMBLED HYPOTHETICAL PROGRAM

PG	LOC	MACHINE CODE			LABELS	MNEMONICS	COMMENTS
000	121	340	000	110		JOZ TEMPIN	If just right, form steady loop.
000	124	320	000	100		JFS START	If too hot, increase CW flow
000	127	123	000	002		LDI INDEX,000 002	If too cold, increase HW flow
000	132	017				LDX A	By fetching current HW value
000	133	107				INR A	To Accum & increasing it
000	134	333	002			OUT PORT2	Send new value to valve
000	136	117				STX A	And save new value in memory
000	137	304	000	110		JMP TEMPIN	Continue monitoring ops
000	142	123	000	001	FULL,	LDI INDEX,000 001	Set pntr to CW valve setting
000	145	377				CLR A	"Clear" Accum to all zeros
000	146	117				STX A	Store zero in CW storage loc
000	147	333	001			OUT PORT1	Send 0 to shut off CW flow
000	151	127				INR INDEX	Increment pointer value
000	152	117				STX A	Store zero in HW storage loc
000	153	333	002			OUT PORT2	Send 0 to shut off HW flow
000	155	000				STP	Halt computer operation

COMPLETELY ASSEMBLED HYPOTHETICAL PROGRAM

The reader may have noticed that there is considerable pencil work involved with developing computer programs at the machine language level. Much of this type of work is associated with having to write down the lists of machine instructions (first in their mnemonic form) along with associated labels and comments and then writing down the actual machine codes (as they are looked up) and addresses of labels during the "assembly" or translation process. This type of work can become quite tedious when large programs are being developed. The amount of work involved in such cases is often greatly increased in practice because of the human tendency to make mistakes during the various development processes. For instance, forgetting to include a necessary instruction (or, more likely, not realizing the need for a particular directive or series of instructions at the time of program creation) can wreak havoc for a machine language level programmer. Upon finding such a mistake, when using the pencil and paper methods as described in the example, one must do a lot of correcting. For instance, the simple corrective measure of having to insert a single instruction at the midpoint of a worksheet may require erasing the remainder of the instructions on the sheet and then re-writing those directives in order to make room for the insertion. If the error is detected after the assembly process has been performed, then even more work must be done over. This is especially so if any portion(s) of the routine(s) moved were referenced (used as "labeled" points) by other parts of the program as the addresses or locations in memory where they will actually reside will have changed! The seriousness of the problem being discussed probably cannot be truly understood by the reader until he or she actually has obtained some first-hand experience in the discipline.

Fortunately, several aids in the form of programs themselves have been developed over the years to assist programmers develop machine language level programs. These programs alleviate the consequences of having to make changes in developing programs. The programs use the computer itself to help the programmer in the task of further programming the computer.

One such program typically used to ease the task of preparing programs is called an EDITOR program. An Editor program enables a computer to assist people in the task of preparing and manipulating the mnemonic listing or SOURCE CODE for a program under

development. First of all, an Editor program allows a person to use a portion of the computer's memory banks as a text storage area (instead of having to write information on a piece of paper such as a worksheet). Mnemonic listings, complete with labels and comments if desired, may be entered via some sort of input device (such as a keyboard or other type of input device as will be described in a later chapter) into a "data" area in the computer's memory. That is, instead of physically writing a list of instructions on a piece of paper, the list is "written" into the computer's memory. The information stored in the computer is kept in an organized fashion. This is typically accomplished by assigning "virtual" line numbers ("virtual" meaning implied but not actually existing) to text so that specific sections (lines) may be easily referenced by the operator. An Editor program generally has a variety of commands available to the operator that allow information stored in the computer's memory (text buffer area) to be manipulated. For instance, lines of information within the storage area may typically be added, deleted, inserted in between other lines, moved about, changed and so forth. Often the program allows changes within a line (such as spelling corrections or word additions/deletions) to be made. Naturally, the information contained in the text buffer area within the computer may be displayed for review by the operator on an output device such as a printer or cathode-ray-tube (as will be discussed in a later chapter). Using an Editor program a programmer can rapidly create a source listing and modify it as necessary as a program is being mentally developed. When the program is completed a permanent "copy" of the information stored in the computer's memory may be "saved" on a medium such as paper tape or magnetic tape (as will be discussed later). The information saved in such a manner may later be recalled (loaded) back into the computer's memory if desired. An important reason for providing this type of capability is so that if it is ever necessary to make changes to a source listing then the original listing may be rapidly reloaded back into the computer. Alterations may then be made rapidly using the Editor program and a new updated listing obtained in a fraction of the amount of time it might require to accomplish such a task using pencil and paper methods.

Another reason for making a permanent copy of the source listing created when using an Editor program is because another program

called an ASSEMBLER (got any ideas where that name came from?) may be used to process the source listing and (has the reader guessed it?) translate the mnemonics and labels to machine codes and addresses! The reader is undoubtably not at all surprised to realize that since the process of translating mnemonics to machine code is simply a matter of looking up the appropriate code in a table, that the process can readily be done by the computer itself. One just places a large table containing the mnemonics being utilized along with their equivalent codes in the memory of the computer and devises a suitable algorithm to have the computer look up information in the table as it processes each mnemonic in a source listing. As this task is being handled, the computer can also readily keep track of how many bytes of memory are required by each type of instruction and using this information it can be directed to keep a record of the addresses of all "labels" in a routine being "assembled" (in just the same fashion as a human programmer might approach the job). As the addresses of labels used in a program are determined the computer can assign the appropriate values to the address-requiring bytes in various types of instructions. The actual details of the operation of an Assembler program are a little more complicated than indicated here, but most readers should readily see that the concept appears (and is) most feasible. Readers who become heavily involved in the activity of programming at the machine language level will soon find themselves utilizing the capabilities of Editor and Assembler programs. They considerably ease the more tedious aspects of developing computer programs at this most fundamental programming level.

The example program describing the control of hot and cold water faucets assumed that after it had been developed and assembled, it could be loaded into the memory of the computer and immediately executed to cause the computer to perform the desired operations. In practice, such a bold assumption is usually invalid. Seldom is it possible to develop a machine language program of any practical size, and find that it will cause the computer to operate precisely as anticipated on the very first time it is tried. Why not? Because the complexity of many tasks, and the number of steps required to convert algorithms to machine code often results in errors being made during the translation process. Sometimes the program developer will not detect an error in an algorithm being implemented. Or, one may not

notice a translation error if mnemonics are being translated t machine codes by hand. Other types of clerical errors can (and do! occur to render first drafts of programs inoperative. Because thi situation occurs so frequently, programmers generally plan on havin to test and "debug" a program after it has been developed (rathe than simply assuming that it will work just as planned upon initia installation as a program in the computer).

"Debugging" a program, as it is often referred to, consists of veri fying the proper operation of the program under controlled test con ditions. Quite often the process consists of simply exercising portion (for instance, subroutines) of a program one at a time, and ascer taining that each section works on an individual basis, and then i proper harmony with associated routines. Sometimes this process i accomplished by walking the computer through each instructio using a special operational mode available on many computer sys tems called the "step" mode. In the step mode, the computer simpl executes one instruction at a time, and then pauses to allow a operator to check the status of various CPU registers or the content of memory locations.

Debugging operations (particularly on small systems) are some times controlled by the operator viewing display lights and operatin individual switches on the front panel of an appropriately equippe computer. However, this method, as with hand assembly processes, i quite slow and tedious. If a fair amount of this type of work must b done, programmers generally prefer to use the computer itself t help in the test and analysis of a program being developed usin another type of program called a "Monitor" or debugging progran A Monitor program generally consists of a relatively small group o routines that occupy from as little as about 256 bytes of memor on up to as much as 1024 or 2048 bytes of memory (depending o the total capabilities). Quite often these routines are provided in special type of memory that cannot be altered by the computer use called a "read-only-memory." (The feature of non-alterability protect the program from destruction by inexperienced users, and als generally has the attribute that it is not affected by loss of power i the computer system. This means the program in the special no alterable memory may be utilized as soon as power is applied to th system. Regular memory provided in computers does not retain it

contents when power is removed from the system, and will be found in essentially random states when power is initially applied to the system.) The group of routines provided as part of a typical Monitor package enable a system user to conveniently perform the following types of operations using an easily operated input device, such as an alphanumeric keyboard, and a display device, such as an electro-mechanical printer:

> Examine and modify the contents of individual locations in the memory banks of the computer system.
>
> Display or "dump" the contents of sections of memory for operator review.
>
> Display and modify the contents of various CPU registers.
>
> Begin execution of a user program at a specified address.
>
> Set a "trap" or "breakpoint" in memory that will cause control of the computer system to return to the monitor program when encountered.
>
> Fill sections of memory with specified binary patterns.
>
> Load sections of memory from an external "bulk storage" device (see later chapter on I/O devices).
>
> Transfer or "save" the contents of sections of memory onto an external bulk storage device.

Using the capabilities of a Monitor program, a programmer may proceed to check out the operation of a program under development to verify proper operation. If problems (bugs) are detected, the

special features of a Monitor program allow the operator to quickly investigate various portions of the program under study to isolate and correct problem sections.

A Monitor program is another example of a program that uses the capabilities of the computer itself to make actual use of the machine easier for the average operator. This concept of installing one type of a program in a computer to make further programming possible at a more advanced or more convenient level leads to a level of programming referred to as high level language programming. This type of programming is discussed in the next chapter.

PROGRAMMING - USING A HIGHER LEVEL LANGUAGE

If everyone that wanted to use a microcomputer had to use machine language programming (as described in the previous chapter), the number of computer users would be considerably reduced. Truth is, to become an efficient machine language programmer it takes a lot of practice, a lot of patience, in many applications special knowledge, and always more time than most people wish to devote to it. Most (not all!) professional machine language programmers have engineering or mathematical training. Many are highly educated at the college level. (After all, before one can program a computer to do advanced tricks, one has to understand the tricks themselves!)

Fortunately for us, some advanced machine language programmers have spent a considerable amount of time and effort to make computers easy to use by practically anyone. How did they do this? They used their machine language programming skills to develop programs (in machine language) that would enable people to communicate with a computer and direct its operation using simple, everyday words and mathematical symbols. That is, a program is developed in machine language that will allow a computer to interpret directions given to it in a HIGHER LEVEL LANGUAGE. This higher level language is structured so that it is easily learned and understood by the average person interested in using a computer.

There are several important things for the reader to keep in mind concerning high level languages. First of all, one must realize that when a computer is communicating with an operator via a high level language (that is, an operator is directing the computer to perform operations such as solving mathematical problems by programming it using a high level language), then the computer itself must already contain in its memory a program (developed originally in machine language and residing in memory in that form). This program controls the operation of the computer system as viewed by the person communicating with the computer using the high level language. One significant aspect of this that may be immediately apparent is that it requires more memory in a computer to have it perform tasks using a high level language than it would require to do the same task if it

was directly programmed using machine code. This is because the computer must reserve a substantual amount of memory in which the interpretive high level program itself may reside before any allocation is made for storage of the user's program.

Second, the reader should realize that there are a variety of so-called higher level languages available for use on various microcomputer systems today. The differences between these languages relate to their intended use and are characterized by differences in their syntax structure, the types of symbols they may recognize, operations they can perform, and so forth. Some of the names readers may have heard of or seen for such higher level languages include BASIC, FORTRAN, ALGOL, PL/M, COBOL, and APL. Languages bearing those names have been standardized by national and international organizations with the intent that users who learn the syntax of a particular language should be able to utilize the same language regardless of what manufacturer's machine it is implemented on. In practice, this comes down to whether the machine language programmers of a given organization are able to implement the precise "standardized" capabilities of a language on the computer(s) produced by that firm. It is also worthwhile pointing out that while there are a number of standardized languages, it is possible to create entirely new languages by simply defining new syntax, new meanings for symbols, new types of operations, etc., and installing appropriate machine language routines in the memory of the computer that will appropriately decode the higher level directives.

The real purpose behind high level languages is to relieve the computer user from having to be concerned with all the intricacies of machine language programming. The need to be concerned with the mnemonics for individual machine language instructions, the related binary codes, memory addressing, and so forth is entirely eliminated. Gone too is the necessity for being concerned with creating individual algorithms to perform simple procedures such as adding two multi-byte numbers together.

Using a high level language, the computer user can concentrate simply on the problem to be solved. For instance, if one wants to add two decimal numbers together, and then multiply the result of the addition operation by a third number, one need simply type a

statement such as:

LET R = (A + B) x C

on a suitable input device connected to the computer. The high level language program in the computer's memory would calculate the value of 'R' given that 'A,' 'B,' and 'C' had been defined previously. (Directing the computer to perform a similar operation might require hundreds, even a thousand or more individual machine language instructions, particularly if A, B, and C were representing, say, ten-digit decimal-based numbers.)

Another important benefit of utilizing a standardized high level language is that it allows users of different manufacturers machines to be able to exchange programs and communicate with one another using a common vernacular. Remember there are a variety of CPU devices available on the market, all of which have varying degrees of differences in their instruction sets and general capabilities. The actual machine code to perform any specific operation will be different for each type of system that uses a different CPU. Thus, it becomes quite difficult for a person knowing the actual machine codes for one type of machine to communicate the specific machine language level steps required to perform an operation to a person familiar with the machine code for another type of machine, when the latter person is not familiar with the machine code of the originator's machine. The problem is analogous to a person trying to communicate in English to someone who only speaks, say, French. The employment of a high level language is analogous to the use of mathematical symbols which have a considerable amount of commonality in any language, being understood by those who speak English and those who converse in French. This will become quite clear to the reader as examples of the capability and use of a typical high level language are presented in this chapter.

High level languages for microprocessor based systems are generally implemented in a fashion referred to as "interpretive." However, there is another manner in which they may be implemented, particularly if a system is supported with fast external bulk storage memory devices. (To be discussed in the next chapter. Briefly, such devices can store "copies" of the contents of a computer's memory

and restore the same information back into memory upon command. Thus, they may be used to save programs and/or data used by a computer.) Systems equipped with fast external bulk storage devices are suited to run high level languages configured in a "compile, then operate" arrangement, and are referred to as "compiler" oriented languages or versions of languages. The differences between the two fundamental configurations will be discussed briefly here for several reasons. First, for a number of technical reasons, the "interpretive" arrangement is currently the vogue amongst microcomputer users. However, with continued improvements in I/O devices and external (fast) bulk storage devices, it is possible that in the future compiler oriented versions will become considerably more popular. The reader should be aware of the current trend and the potential for change. Second, the reader should be aware of the advantages and disadvantages to the two methods that may have bearing on what type of equipment one might lean towards obtaining as a microcomputer system is acquired.

Both types of higher level languages process STATEMENTS made by the user. Usually, a user develops a high level program by preparing a whole series of lines of information containing statements that tell the computer how to proceed in order to solve a particular problem or control a process. Statement lines typically consist of a KEYWORD that defines the type of operation to be performed along with numbers, symbols, and operators that support or may be utilized during the operation. A typical statement line was illustrated earlier in the form:

LET R = (A + B) x C

In the example statement line, the keyword LET indicates that a "set equal to" type of operation is to be performed. The character R that immediately follows the keyword LET serves as the symbolic representation of a variable value. The mathematical expression to the right of the equal sign in the statement line indicates the value that the symbol R is to be set equal to. In the example line, the characters A, B, and C represent symbolic names for numeric values (that would normally be defined by other statement lines before the statement was processed).

When developing a high level language program, each line containing a statement is preceeded by a number to provide a way for the programmer as well as the computer to refer to a particular statement line. The number serves the same purpose that a label does when developing a machine language program using an assembler. It identifies (serves as an address) a particular instruction point in a program. (Note that each statement line in a high level language program may be considered as analogous to an instruction or directive in machine language except that it is many times more powerful, and in fact, will invoke the performance of typically hundreds or even thousands of machine language instructions within the microcomputer.)

Naturally, this procedure of creating a series of statements would be performed after the programmer had defined the problem to be solved. Typically one would also create a flow chart of the operations necessary to reach a solution, as is done when preparing machine or assembler language programs.

If a compiler type higher level language was to be used by the computer operator, the statement lines written in the higher level language syntax would typically be prepared using a separate EDITOR program just as may be done when preparing a machine language program using mnemonics. (Remember, an Editor program simply serves to assist a person prepare text of any kind, regardless of what its end use will be!) If an interpretive type higher level language was being utilized, the preparation of the statement lines (which are often referred to as the HIGH LEVEL SOURCE CODE) would be accomplished through the use of an editing type of routine provided as part of the overall interpreter! At first thought, a newcomer might be asking, "so what is the big deal here?" The first big deal is that an interpretive language processor must have editing capability built into it, while a compiler type language considers preparation of the statements on which it is to operate as an entirely separate operation. This means that an interpretive language is going to require more memory in the computer - it has to have room for the editing routines! It will also have to provide room for all the lines of information contained in the statement lines! (After all, that is what is manipulated by the editing portion of the interpreter.) However, a compiler type language typically processes higher level statements

(HIGH LEVEL SOURCE CODE) as they are "fed in" from an external bulk storage device; hence, computer memory space does not have to be dedicated for such purposes.

Once the high level source code (statemei lines) have been prepared for a program, a compiler oriented high level language would operate in two distinct parts as follows: First the statements would be read by a special program called a COMPILER, and those high level statements converted to sequences of machine language instructions that would perform the desired operations. Typically, the machine language instructions prepared by the compiler would need to reference an array of special RUN TIME routines that would always be available in the system for which a program was being compiled. These RUN TIME routines would typically be structured to handle commonly used mathematical operations, conversions, and so forth. The sequences of instructions produced by the compiler would eventually be coupled to the RUN TIME routines and controlled for final execution by a separate section of a compiler oriented language called an EXECUTIVE program.

In an interpretive language, the high level statements prepared by the programmer - instead of being converted to actual sequences of machine language instructions - are immediately "INTERPRETED" and executed by calling upon appropriate subroutines contained within the interpreter program. (Those subroutines are themselves made up of machine language coded directives.)

To provide further clarification of the major differences between the two types of high level languages, consider the example statement line:

LET R = (A + B) x C

A high level language arranged as an interpreter would process the line in the following manner: Upon scanning the line and recognizing the key statement type indicator LET, it would "realize" that it was to set the value of the variable named on the left side of the equal sign to the value of the expression found on the right side of the equal sign. It would then proceed to find the name of the variable being defined (R) in the line, and save space for the variable

name (R) and its eventual value in memory within the computer. The interpreter would then proceed in an orderly fashion to analyze the expression to the right of the equal sign in order to immediately determine an actual value for the variable R. In other words, it would immediately proceed, by calling upon appropriate addition and multiplication subroutines provided as part of the interpretive program, to look up (in an appropriate variables table) the values of the symbols A, B, and C, add the value for A to that for B, and multiply the result by C to obtain the final value for R. When that value had been calculated, it would be stored in the place reserved for it in the memory of the computer. At that point, all of the information in the statement line would have been acted upon. The interpretive program would then be ready to go on to any additional statement lines that might be provided in a series of such statements making up a high level language program.

On the other hand, a compiler oriented high level language would process the line of information in the following manner: When it scanned the line and detected the LET statement directive, it would proceed to start generating machine language directives (code) that when in turn were executed would result in the desired operations being performed. Thus, machine code instructions would be produced that would cause a variable value (R) to be set equal to the value obtained when machine code directives (produced by the compiler) capable of looking up the values of variables named A, B, and C and performing the indicated addition and multiplication operations were executed. Thus, the output of a compiler is machine language code. The machine language code produced by the compiler when finally executed actually solves the problem. In order to simplify the process somewhat, the compiler is typically designed so that machine language sequences it produces can call upon the special "run time" routines typically provided as part of a compiler oriented language. Thus, every time a standard operation such as "multiply one number by another" is encountered, it is not necessary for the compiler to generate a long string of machine language directives for that purpose. Instead, it merely generates the machine code instruction to CALL a particular "run time" subroutine that it knows will be available within the computer's memory when the machine code produced by the compiler is actually executed.

In essence then, a compiler oriented language produces machine language code that may later be executed to obtain the desired solution to a problem. An interpretive language does not produce machine code as an intermediate step, it immediately processes information to obtain the sought after answer.

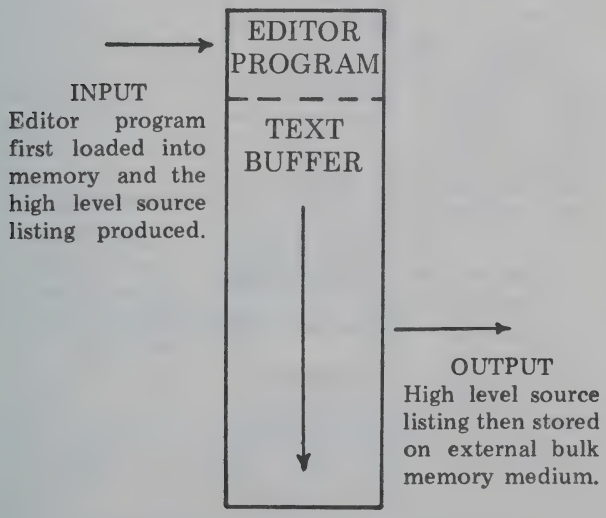
At first thought one might be tempted to say, "Who needs a compiler oriented language?" After all, it has been mentioned that the method requires a separate Editor program with which to prepare the source code (high level program). Additionally, the compiler essentially operates in two parts. One part produces machine code, the other part consists of "run time" routines and a program that controls the execution of the machine language routines produced by the first part. It may also be apparent that this multi-part procedure is going to place an extra burden on the microcomputer system in regards to input/output devices, and external bulk storage devices. These devices are needed to hold the results of intermediate steps and provide for efficient loading and unloading of the various sections of the compiler system to/from the computer.

Compared to the compiler, the interpretive language seems to win "hands down" at first glance. It appears so convenient. Just enter a program and have it produce the answer! But at what cost this convenience? Memory! As a general rule, an interpretive language requires more memory in the host computer. This is because the interpretive program, including its Editor section, along with the high level user's program (the lines of statements that make up the high level program), and all of the routines necessary to perform any of the operations that the interpreter can indeed recognize and accomplish, must all reside in the microcomputer's memory banks at one time! It takes more memory to run interpretive languages, and memory is expensive.

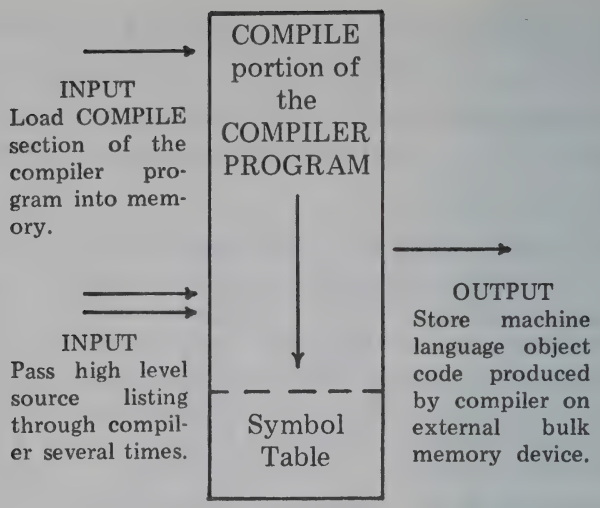
Anything else negative about an interpretive language? Yes. As a general rule of thumb, it is slower in operation because it is constantly having to "interpret" a statement line before it performs an operation. In programs that perform a lot of repetitive operations (program loops), this factor can greatly decrease the speed at which various types of problems may be solved. Remember, a compiler type language only has to "decode" the original line of high level

source code (a statement line) once. It converts the statement type to a series of machine language instructions that will actually perform the operation. When it comes time to execute those instructions under "run time" control of the compiler executive, there is no "interpretation" necessary. Repetitive operations are thus performed much more rapidly.

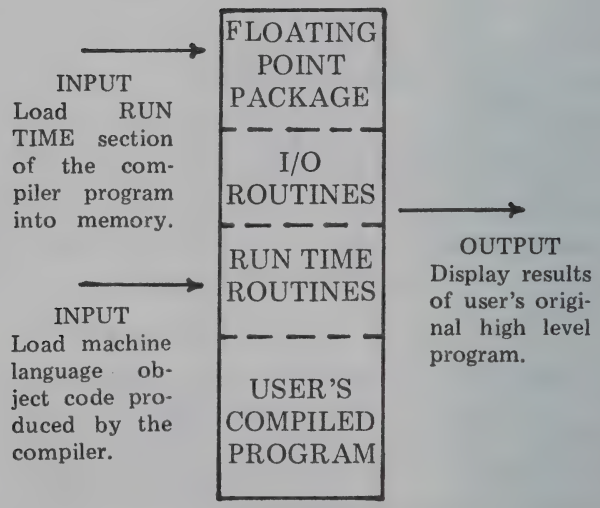
Despite the requirement for extra memory and the relative slow speed at which it operates, the current most popular method for implementing a high level language is the interpretive method. The convenience features of this method, given the current state of the art in the area of low cost, reliable, and fast external bulk storage devices (which are not sufficient to adequately support compiler type languages in the price range that most individual microsystem owners would find attractive), outweigh the positive attributes of the compiler languages in the opinion of the majority of users at the present time. (As hinted at, however, this may some day change as technology pushes forward towards better, faster, lower cost peripheral devices of the type needed to adequately support practical compiler oriented languages.)



COMPILER OPERATION - STEP 1

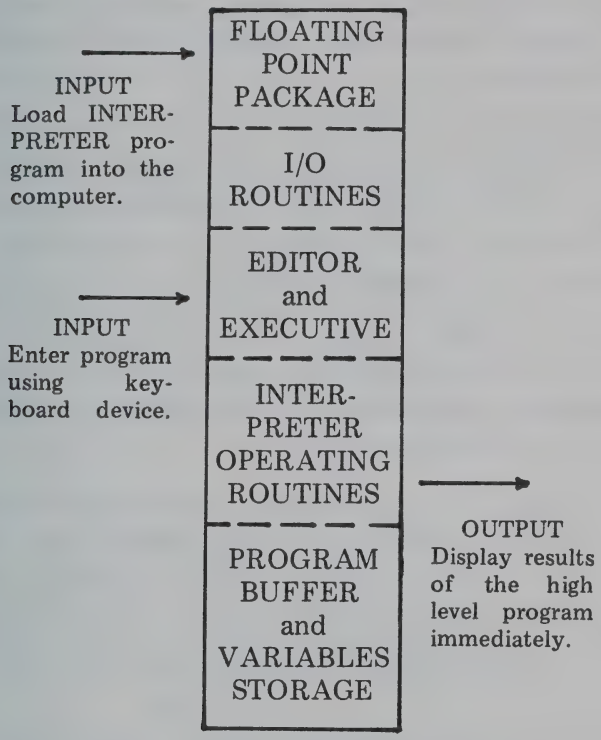


COMPILER OPERATION - STEP 2



COMPILER OPERATION - STEP 3

The accompanying diagrams summarize the steps typically necessary to execute a higher level program using first a compiler oriented language, and then an interpretive language. Note the drastic decrease in the amount of I/O operations required of the latter method. The diagrams also depict how memory is typically assigned as a function of which type is being implemented illustrating the increased requirement for memory in the interpretive language.



INTERPRETER OPERATION

The reader who after reading the previous chapter has decided that machine language programming may not be quite the level at which one wants to deal with a computer, should at least prepare to learn something about programming using a higher level language if he or she has any notion of utilizing a microprocessor system effectively.

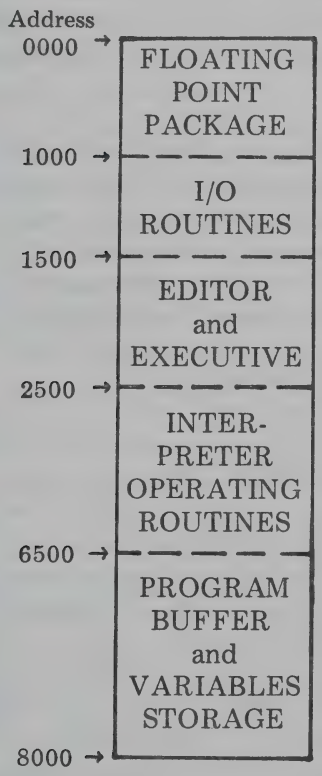
The ability for the machine to be programmed at a person's will is what gives it advantages over other types of machines. If the machine operator does not learn to program a computer at some level, there is no need to have any interest in the machine. Programmability is its essence. Not being able to program it leaves one at the mercy of "canned" program producers who can never provide the individual with all the features one might desire. Fortunately, using a high level language, programming a computer can become so easy, so far removed from the tedious types of aspects mentioned in the previous chapter, that virtually anyone can learn to tell the computer how to perform useful tasks!

Since interpretive type high level languages are most popular, and most readily available for microprocessor systems, an interpretive type language will be illustrated to show how easy programming in a higher level language can be (especially compared to machine language programming as explained in the previous chapter.) Remember, though, as you read the following, that working with a high level language is made possible because somebody else took the time to work out all the routines required to implement the language using machine language programming methods. (That is no small task, typically requiring several man years!)

Without question, the most popular type of higher level language currently in use by microprocessor enthusiasts goes under the acronym BASIC. (The acronym BASIC incidently, stands for Beginners All-purpose Symbolic Instruction Code.) Since it is so popular, the following examples will be based on that language to illustrate how a typical high level language appears to a user.

In order to utilize an interpretive higher level language such as BASIC, it is first necessary to install the interpreter program itself in the memory of the computer system on which it will be run. While some small versions of limited BASIC will reside and operate in about 4000 bytes of memory, more common versions of the language require a system having at least 8000 or more words of memory. If one remembers what was said earlier, about the fact that an interpretive language requires that the higher level language program that a user desires to execute must be resident in memory along with the interpreter, one may deduce that the interpreter program

self must not take up all of the available memory in a system. Some pace must be left over in which to store the user's high level program! In many instances, the interpreter is arranged such that all vailable memory within a system beyond a certain address value nay be used for storage of a user's high level program. Thus, a ystem having 8000 words of memory, 6500 words of which were sed by the interpreter program, would provide 1500 bytes for torage of a user's program that could be processed by the interreter. Suppose that the same system was expanded so that it conained 12,000 bytes of memory? The computer would then have oom for 5500 bytes of high level source code when the interpreter vas in memory. The accompanying diagram illustrates how an inter-



Example Memory Map
of a BASIC Interpreter

pretive program such as BASIC might be mapped into the memory of a computer system. Notice that the interpreter program itself may be considered as consisting of several major sections. These sections typically include: An editing portion that enables a user to enter the source code for a high level language program into the "user program buffer area." An executive portion that allows the operator to switch between editing operations and execution of the stored high level program. The interpreter routines themselves that actually decode the high level syntax statements and perform the associated operations. And, a number of general routines to handle various mathematical operations and functions, control input/output operations, and so forth. Finally, the diagram indicates the remainder of available memory being used to hold the high level statement lines that form a high level program. Those statement lines are placed there under control of the Editor portion of the interpreter as directed by the computer user.

Once the BASIC interpreter has been loaded into the computer's memory (using equipment that will be discussed in the next chapter), a computer user is able to control the overall operation of the interpreter, usually by entering directives on a device such as an electronic keyboard. The status of the interpreter and the results of operations may be observed on a suitable display device connected to the computer, such as an electro-mechanical printer, or a cathode-ray-tube (television-like), or similar display.

Directives that control the overall operation of the interpreter are referred to as COMMANDS. What are some typical commands? SCRATCH is a typical command! This directive in BASIC language is used when the user wants to start building up a new high level language program in the program buffer (using the editing capabilities built into the interpreter). The command SCRATCH directs the computer to effectively clear out all the previous statement lines stored in the program buffer area, and any associated data of information associated with variable names. (Remember, the user program buffer area is that section in memory used to store a high level program written in the syntax of the high level language.)

How do lines containing syntax get placed into the program buffer area in memory? They are placed there by the editing portion of the

interpreter program. In most BASIC interpreters, whenever a line containing a statement is preceeded by a number, it means the statement line is to be stored in the high level program buffer to become part of a series of statements that will make up a high level language program. In fact, the numbering of lines by the programmer is used to order the statement lines into the sequence in which they will normally be executed. The editing portion of the typical interpreter operates in such a fashion so as to keep the lines of information ordered within the program buffer according to the value of the number preceeding each statement line. Thus, if the user program buffer currently contains lines numbered 90, 95, and 105, and the operator decides to add a statement line preceeded by the number 100, that line of source code will actually be placed in the program buffer so that it resides between the lines already in the buffer that start with the numbers 95 and 105 as illustrated in the accompanying diagram. (This is actually accomplished by "moving" any lines numbered above 100 further up in the buffer thereby providing space for the new line to be inserted.) Remember, too, in high level programming, a line number serves as an effective "address label" by which to refer to a particular high level instruction. (This is analogous to the technique of assigning a symbolic name in place of an actual address value when developing programs using assembly language methods as discussed in the previous chapter.)

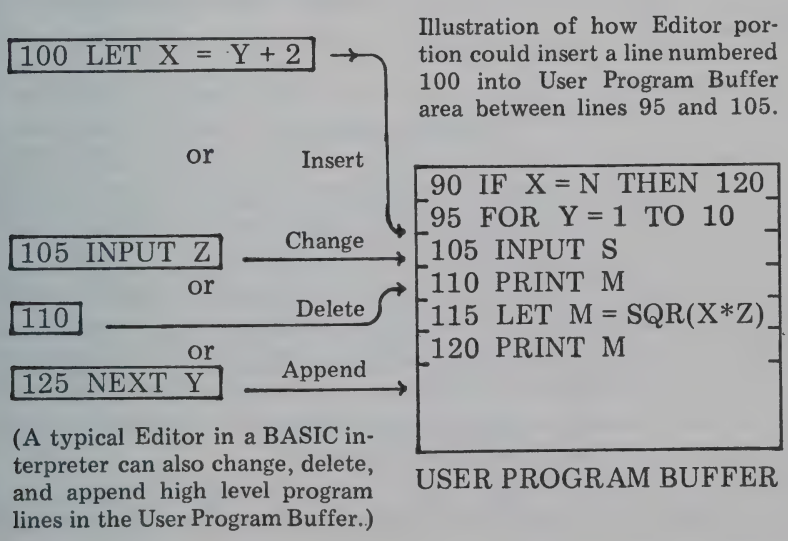


Illustration of how Editor portion could insert a line numbered 100 into User Program Buffer area between lines 95 and 105.

(A typical Editor in a BASIC interpreter can also change, delete, and append high level program lines in the User Program Buffer.)

Thus, to build up a high level program to be executed by the computer, the operator simply types in numbered lines containing statement types arranged in an order the programmer feels appropriate for the particular type of problem being solved. These lines will be stored by the editing portion of the interpreter in the user's program buffer area in memory. When the operator desires to review the current contents of the program area in memory, another executive command called LIST may be used. The LIST command simply tells the interpreter to list out all the lines stored in the user program storage area for review by the operator!

The RUN command, as the reader might well immediately deduce, directs the interpreter to commence interpreting and executing the high level program contained in the program storage area.

Wow! What simplicity! The user, using three simple commands, can clear out the program storage area to get rid of an old program (SCRATCH command); enter a new series of numbered statements to form a new program, and LIST the statements for review; and then issue the RUN command to have the new program executed. That is a far cry from machine language programming! Is it really that simple to use a high level language such as BASIC? Well, just about - read on to learn how a high level program may be created using the various types of STATEMENTS available in BASIC language.

STATEMENTS, as referred to in BASIC language, are lines that contain special KEYWORDS indicating the types of operations that are to be performed by the computer. In addition to a KEYWORD, which normally immediately follows the line number that prefixes a line, a statement line will contain supporting information, such as data that is to be operated on or mathematical expressions. The reader has already been introduced to one type of BASIC statement in several examples in this chapter. That is the LET statement which is used to indicate the value to which a variable symbol is to be set. The following discussion introduces a number of the common statement types typically found in high level BASIC interpreters, and the manner in which they typically are used.

The reader has already been introduced to the LET type of state-

ment. It is certainly a "workhorse" statement in BASIC language, as virtually any program that needs to solve mathematical formulas will utilize the LET statement. It is important to point out at this juncture that the LET statement, and many other statement types in the language, typically contain formulas that contain mathematical "operator" symbols, such as the "+" and "-" signs, and the slash (/) sign typically used to indicate division. When it comes to signifying multiplication, instead of using an "X" (as shown in the previous examples), an asterisk "*" sign is generally used in BASIC to avoid confusing the indicator for multiplying with the alphabetical character "X." (This procedure also allows the letter "X" to be used as a variable name.) Also, to simplify exponentiation (raising a number to a power), a double asterisk (**) or an up-arrow "↑" is typically used. This is because most computer display systems have difficulty displaying an offset value to indicate exponentiation as is normally done when writing formulas using manual methods. (Such as, for example, 14.2^3 indicating raising 14.2 to the third power.)

When mathematical operators are used in a statement line, they are always processed by a consistent set of rules so that all expressions will be evaluated in the same manner. For instance, in BASIC, an order of precedence (heirarchy) is given to mathematical operators such that exponentiation is always performed ahead of multiplication or division. Those operators in turn are processed before addition or subtraction operators. This type of "rule-making" insures that an expression such as:

LET Z = 3*2↑2-4

is always interpreted so that the answer is: 2 raised to the 2'nd power (which equals 4), times 3 (cumulative answer is now 4 times 3, which is 12), minus 4 (final answer is 12 - 4 = 8). If such an operator precedence system were not established (and adhered to by the high level language programmer), a person and a computer might come up with different answers for the same formula. For instance, if

LET Z = 3*2↑2-4

was calculated simply by scanning from left to right, one would obtain the value 32 as an answer instead of 8! (3 times 2 would be 6.

6 raised to the 2'nd power would be 36. Take away 4 from 36 for an answer of 32!)

Note too, that when using BASIC, unknown values (values that are to be solved for in a LET expression, etc.) are called variables. A variable is signified by assigning a symbolic letter, or several letters, or a combination of letters and numbers (beginning with a letter) in place of a numeric value. For instance, "A," "BB," and "C1" are all examples of variable names in BASIC.

In summary, the LET statement provides a simple way for the operator to direct the computer to solve a mathematical formula. The next type of statement to be discussed is sort of an extension of the LET statement. It is called the IF. . . .THEN statement, and it allows the user to direct a high level program to make a logical decision based on the value of an expression at the particular time it is evaluated. For instance, the LET example previously demonstrated could be altered to form an IF. . . .THEN statement having the following format:

```
IF X = (A + B) * C THEN [NEW STATEMENT]
```

This type of statement would be interpreted in the following manner: If the present value of the variable "X" is exactly equal to the value of variables "A" plus "B" multiplied by "C," then the computer is to perform the NEW STATEMENT that immediately follows the THEN directive on the same statement line. If the value of "X" is not exactly equal to the quantity in the expression, the THEN directive is to be ignored! (In that case, the computer immediately proceeds to interpret the next higher-numbered statement line in the program.)

The IF. . . .THEN type of statement allows comparisons between quantities to be made. The comparisons do not have to be made strictly to determine equality. BASIC allows a combination of tests to be made in an IF. . . .THEN type statement, such as "less than" or "greater than" with or without the test for equality. Thus, one may test to see if a variable is:

less than
greater than
less than or equal to
greater than or equal to
less than or greater than
equal to

It will also be pointed out that the IF. . . .THEN statement also allows complete expressions to be used on the left hand side of the conditional indicators so that statements such as:

IF X + 2 = Y - 4 THEN LET Z = X

may be utilized.

The IF. . . .THEN statement is frequently coupled with another type of statement called the GOTO statement in order to direct a program around groups of instructions when selected conditions are met. The GOTO statement has the format:

GOTO LLL

where "LLL" stands for the "line number" of a statement line. Remember that all statement lines stored in the user program buffer must be given a line number? That line number serves as an effective address that can be referred to in order to identify the particular line by other statements in a high level program. The GOTO statement in BASIC is the high level language equivalent of a JUMP directive in machine language. Now, instead of the program jumping to an absolute memory address, as far as the programmer is concerned, it jumps to a statement identified by a particular line number. The GOTO statement is frequently used to direct a program around a portion of a program that might, for instance, go unused if an IF. . . .THEN statement meets a specified condition. Also, it is often used to direct a program back to a particular starting point in a program requiring multiple execution of a series of statements.

Speaking of multiple executions of sections of a program. . . the need arises so often that, as in machine language programming, the ability to create "subroutines" is highly desirable. BASIC allows the

"calling" of groups of statements to form such subroutines through the GOSUB statement. Just as with the GOTO statement, the GOSUB statement takes the form:

GOSUB LLL

where the "LLL" indicates the line number that prefixes the statement line of the first line in a group of lines that is to be used to form a high level language subroutine.

A subroutine in BASIC language consists of one or more statement lines in a program. The finishing point of a subroutine must be indicated by the statement RETURN.

Whenever a GOSUB statement line is interpreted by BASIC, the line number of the statement line immediately following the GOSUB statement is saved by the computer on a line number stack. (Just as the program counter is saved on a stack when a machine language subroutine is called by referencing an absolute memory address.) The line number saved is the line number to which the program is to return when the statements making up the subroutine have been executed. Therefore, when a RETURN statement is encountered in a BASIC program, the computer will go back to the line indicated by the line number stored at the top of the line number stack!

Most BASIC languages provide a line number stack sufficient in size to allow a goodly number of subroutines to be nested one inside another. Thus, one subroutine may call another, which in turn may call still another, and so forth, to six, seven, eight, and sometimes more levels. The program will still be able to return all the way back to the original starting point when the proper number of RETURN statements are encountered. This allows for the development of highly efficient programs when repetitious operations must be performed.

Repetitious operations are so often necessary for solving many types of problems that BASIC language provides another type of statement specifically designed to facilitate the formation of loops in programs. It really consists of two types of statements - the FOR statement which indicates the starting point in a loop, and the NEXT

statement which indicates the point in a program at which a loop is to be repeated (or terminated).

The FOR statement that initiates a program loop takes the format:

FOR X = N TO M STEP S

where X indicates a variable value which is to be initially equal to N, S is a step size by which X is to be incremented each time the FOR. . . .NEXT program loop is traversed, and M is the final value of X. When the value of X reaches M, it will cause the program to exit the FOR. . . .NEXT loop by causing statement interpretation to continue with the line that immediately follows the loop terminating NEXT statement.

The body of the program loop initiated by a FOR statement line consists of all those statement lines between a FOR statement and its terminating NEXT statement. FOR. . . .NEXT loops are identified by the variable with which they are associated. Thus, to terminate the FOR statement line used for demonstration purposes, the NEXT statement would have to appear as:

NEXT X

with the variable letter "X" corresponding to the variable name used in the loop initiating FOR directive! This procedure of relating or coupling each FOR. . . .NEXT loop by variable name allows FOR. . . .NEXT loops to be nested one inside another - a useful feature in many types of problem solving programs.

It is also important to understand that the variable value associated with a FOR. . . .NEXT loop is incremented by the STEP size indicated as part of the FOR statement line each time the loop is traversed from start to finish. Thus, if a FOR. . . .NEXT program loop appeared as follows:

```
50  FOR X = 1 TO 5 STEP 2

60  LET Z = X*X + 2X + 5

70  NEXT X

80  . . .A STATEMENT LINE. . .
```

then the formula for Z would be initially solved with X equal to 1 The second time through the loop it would be solved with X equal to 3. The third time through X would be equal to 5. That would be the last time that line 60 was processed. This is because when X exceeded the upper limit value of 5 specified in the FOR statement line, the program would skip instructions contained in the statement lines between the FOR statement and the NEXT statement, and resume operations with example statement line number 80. That is because line 80 immediately follows the loop terminating NEXT statement labeled line number 70.

The types of statements discussed to this point would do little practical good if the language did not contain several statement types such as the following that allows information to be inputted to a program and displayed by the program after calculations have been made.

The INPUT statement, along with several specialized types related to the inputting of information to the computer (such as READ and DATA), provides BASIC language with the capability of obtaining information from an operator as a program is interpreted. For instance, upon encountering a statement in the form:

```
INPUT X,Y,Z
```

a BASIC program would wait for an operator to enter three values from, for instance, a keyboard input device; and assign those values respectively to the variables named in the statement (X, Y, and Z)

To display information calculated by a BASIC program, the programmer may utilize the PRINT statement. The PRINT statement type may be used in a format such as:

PRINT X

which will simply cause the value of the variable X to be displayed on the system's display device. However, the PRINT directive also allows more complex outputting operations to be performed.

For instance, a special feature of the PRINT statement is that by enclosing text within quotation marks, the programmer may cause that text to be displayed when the PRINT statement is executed. This provides an easy method of identifying information displayed by the computer. For example, the statement line:

PRINT "THE VALUE OF X IS =" ;X

when executed would yield the sentence:

THE VALUE OF X IS = 3.5

if, at the time the statement was encountered, the value of X was 3.5. The semicolon in the PRINT statement line illustrated is typically used to provide formatting information to the interpreter. In this case, it is assumed to mean "print the variable value immediately after the text portion." Similar types of special characters may be used within PRINT statements to provide other types of formatting, such as spacing to selected columns within a line; whether or not to provide additional information on the current display line; or to provide for a new display line on the output device.

To facilitate documenting high level programs, most BASIC interpreters allow the use of a REMARKS statement line. A statement line containing this keyword is effectively ignored by the interpreter. Such lines are used purely to hold information that might be of interest to other programmers who might review or desire to modify the high level program.

An END statement is used to inform the BASIC interpreter that it has reached the end of a program, and is to cease high level program execution. When an END statement is encountered, the interpreter switches from the program execution (RUN) mode to the executive COMMAND mode. The operator may then proceed to request more

runs of the program with new data, or to use the editing portion of the interpreter to modify the program, create a new program, or perform other operations.

While almost all versions of BASIC language available today can interpret the types of statements presented in this section, many versions incorporate additional types of statements that enhance the power and capability of the language. Some versions provide for the manipulation of mathematical matrices by providing so-called DIM (for DIMension) statements. Some provide powerful STRING-HANDLING facilities that enable people to manipulate strings of text. With this capability, powerful operations such as sorting lists of names or other data may be accomplished as easily as one solves mathematical formulas using the computer.

The purpose of this chapter is simply to give the reader a feel for the difference between programming in a high level language versus programming at the machine language level as discussed in the previous chapter. Hence, a continued discussion of all the possible statement types that various versions of BASIC may contain is not in order here. However, the prospective microsystem owner should be aware that individual versions of the language will vary in capability because of the addition of statement types not mentioned in this introductory discussion.

In addition to the major capabilities of interpreting statements such as those explained or mentioned in this chapter, BASIC language interpreters are typically able to process special FUNCTIONS that may appear within statement lines. Functions differ from statements in that they operate on an argument that immediately follows the FUNCTION mnemonic (and is generally enclosed in parenthesis). Some of the kinds of functions that most BASIC interpreters are able to calculate include:

> The SQR (SQuare Root) function which takes the square root of the argument associated with the mnemonic. For instance, specifying:
>
> SQR(4)

in a statement line would result in the value 2 being used as the function value.

The SGN (SiGN) function merely checks the sign of a value and substitutes a -1 if the value is negative, +1 if the value is positive, and 0 otherwise. It is frequently used with a variable or expression as the argument and is useful, for instance, for determining at what point a formula reaches a critical (say, positive or negative) value. For instance, the function:

```
SGN(X*X - 4*X - 4)
```

would return "-1" if X was equal to 4, but return "+1" if X was equal to 5 (or more). Note that the function does not substitute the value of the expression in a statement. It substitutes -1, 0, or +1 as the case may be.

The INT (INTeger) function finds the greatest integer value possible in the value signified in the argument. That is, it takes the greatest whole number less than or equal to the number contained in the argument. For instance:

```
INT(54.999)
```

or

```
INT(54.001)
```

both return the value 54 (54.000 for clarity) since that is the largest WHOLE number that may be contained in the value expressed in the argument of the function. (It is easy to get tripped up using this function with negative numbers. The function yields -2 for instance, if INT(-1.9) is specified because on the conventional numbering scale -2 is smaller than -1.9. Since -1 is larger than -1.9, it does not meet the criteria necessary to satisfy the function that the

unwary sometimes think it will!)

The ABS (ABSolute) function may be used to obtain the magnitude of a number without regard to sign.

Almost all versions of BASIC include a highly useful function that is signified by the mnemonic RND for RaNDom. When the RND function is requested, the interpreter generates a "pseudo-random" number (usually in the range of greater than zero to less than 1.0, such as from .0001 to .9999). This function allows programmers to develop simulations, games of chance, etc.

As is the case with statement types that may be provided in a BASIC language interpreter, beyond a certain minimum point an interpreter developed by one organization may have more (or different) kinds of functions available than an interpreter from another supplier. Advanced interpreters frequently include functions that can calculate sines, cosines, and logarithms to various bases, as well as anti-logs and other kinds of so-called transcendental mathematical functions. One is urged to consider one's own needs, and compare the capabilities of various versions if and when one plans to acquire a language such as BASIC for a microcomputer system.

Now that the reader has an understanding of what types of directives one is able to command a computer to perform using such a language, it is time to illustrate the kind of power these directives have when grouped together in the form of a high level program.

As when doing machine language programming, it is advisable to proceed to develop high level programs in a neat and orderly manner. One can start out with a verbal description and then develop a flow chart. However, because of the nature of high level programs - the fact that statement lines embody whole concepts at once (whereas each instruction at the machine language level represents just a tiny step in a concept or operation), and the fact that the statement lines themselves act as a verbal and mathematical expression of the operation, one will find that flow charts need only provide a broad, conceptual

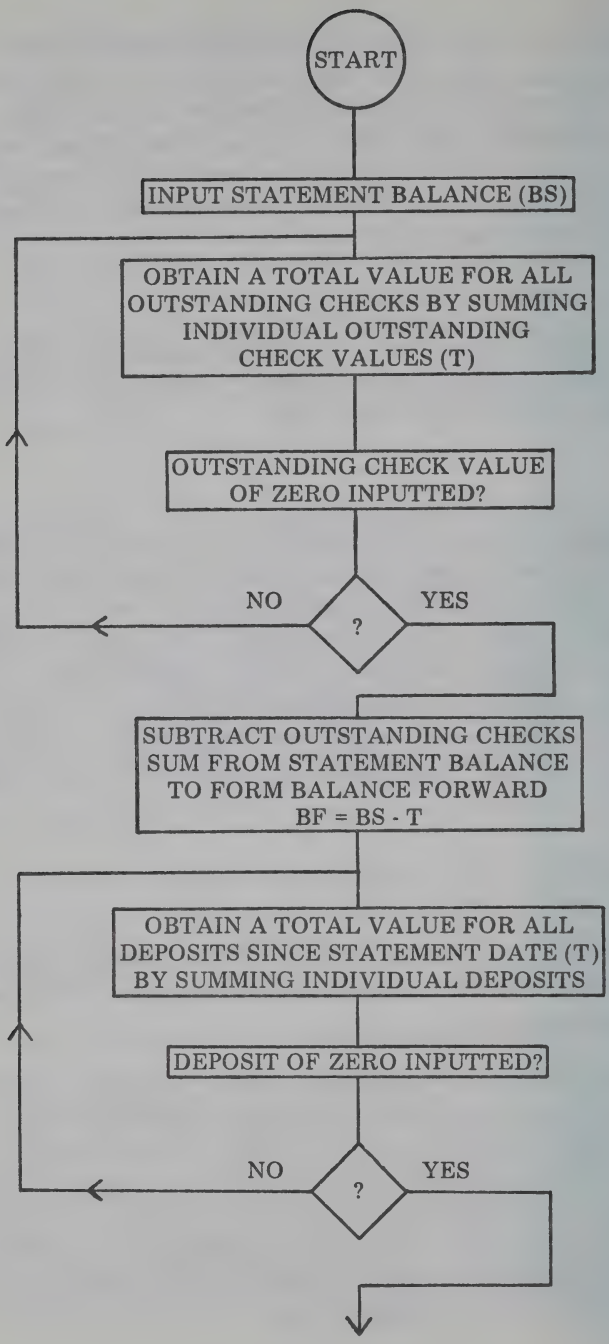
guideline. There is no longer the need for long, detailed, step-by-step kinds of charts associated with machine language programming to take care of such mundane tasks as adding two digits together, etc. Compare the following high level program development example with that of the tank filling program discussed in the previous chapter. The reader can see that a lot more gets done with a lot less work when using a high level language!

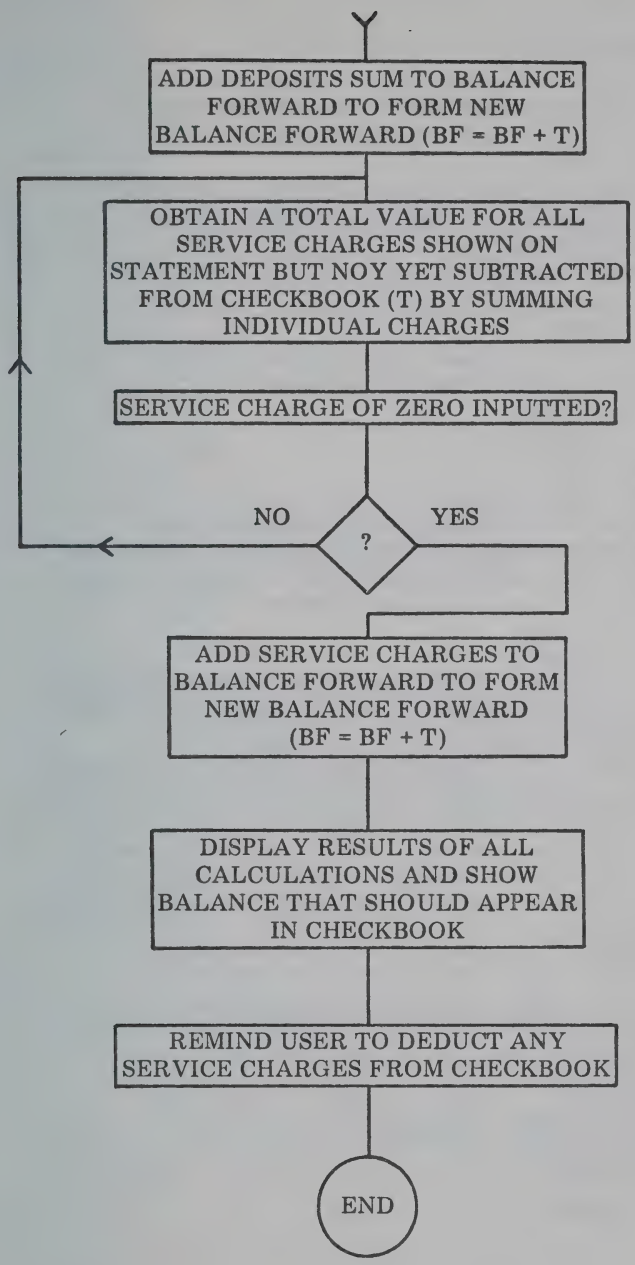
Problem: Develop a program using BASIC language to assist a person balance their checkbook and see that it agrees with a monthly statement received from the person's bank.

Verbal/mathematical proposed solution: Use reported balance as shown on current bank statement as starting point of calculations. Subtract any outstanding checks shown in checkbook from bank statement balance. Add any deposits made since statement date to bank statement balance. Add any service charge levied by bank shown on statement, but not yet subtracted from checkbook to the statement balance. Calculated amount should equal balance shown in checkbook. Update checkbook by subtracting bank service charges if applicable.

A flow chart of the proposed operations illustrates the program in slightly more detail than the verbal description by identifying looping operation where they occur.

The program as it would appear in BASIC language follows the flow chart. Note that, even without using any REMark statements in the program listing (which would be ignored by the computer), it is quite easy to follow the overall operation of the program, especially if one compares it to the accompanying flow chart. The entire procedure is a far cry from developing such a routine using machine language techniques where each tiny operation could take tens to hundreds of individual machine language steps!


START
INPUT STATEMENT BALANCE (BS)
OBTAIN A TOTAL VALUE FOR ALL OUTSTANDING CHECKS BY SUMMING INDIVIDUAL OUTSTANDING CHECK VALUES (T)
OUTSTANDING CHECK VALUE OF ZERO INPUTTED?
NO
?
YES
SUBTRACT OUTSTANDING CHECKS SUM FROM STATEMENT BALANCE TO FORM BALANCE FORWARD BF = BS - T
OBTAIN A TOTAL VALUE FOR ALL DEPOSITS SINCE STATEMENT DATE (T) BY SUMMING INDIVIDUAL DEPOSITS
DEPOSIT OF ZERO INPUTTED?
NO
?
YES


ADD DEPOSITS SUM TO BALANCE FORWARD TO FORM NEW BALANCE FORWARD (BF = BF + T)
OBTAIN A TOTAL VALUE FOR ALL SERVICE CHARGES SHOWN ON STATEMENT BUT NOY YET SUBTRACTED FROM CHECKBOOK (T) BY SUMMING INDIVIDUAL CHARGES
SERVICE CHARGE OF ZERO INPUTTED?
NO
?
YES
ADD SERVICE CHARGES TO BALANCE FORWARD TO FORM NEW BALANCE FORWARD (BF = BF + T)
DISPLAY RESULTS OF ALL CALCULATIONS AND SHOW BALANCE THAT SHOULD APPEAR IN CHECKBOOK
REMIND USER TO DEDUCT ANY SERVICE CHARGES FROM CHECKBOOK
END

```
1000 PRINT
1500 PRINT
2000 PRINT 'WHAT DOES YOUR BANK SHOW AS'
2100 PRINT 'YOUR STATEMENT BALANCE';
2200 INPUT BS
2300 PRINT 'PLEASE LIST INDIVIDUAL AMOUNTS'
2400 PRINT 'OF ALL OUTSTANDING CHECKS:'
2500 GOSUB 9000
2600 LET BF=BS-T
2700 LET N1=N-1
2800 LET T1=T
3100 PRINT 'PLEASE LIST ALL DEPOSITS MADE'
3200 PRINT 'AFTER STATEMENT DATE:'
3300 GOSUB 9000
3400 LET BF=BF+T
3500 LET N2=N-1
3600 LET T2=T
4100 PRINT 'PLEASE LIST ANY BANK SERVICE'
4200 PRINT 'CHARGES SHOWN ON BANK STATEMENT'
4300 PRINT 'WHICH HAVE NOT YET BEEN DEDUCTED'
4400 PRINT 'FROM YOUR CHECKBOOK:'
4500 GOSUB 9000
5000 LET BF=BF+T
5100 PRINT 'YOU LISTED'; N1; 'OUTSTANDING CHECKS'
5200 PRINT 'WHICH TOTALED $'; T1; ',AND REPORTED'
5300 PRINT N2; 'DEPOSITS AMOUNTING TO $'; T2
5400 PRINT 'AS NOT BEING CREDITED ON THE BANKS'
5500 PRINT 'STATEMENT. YOUR CHECKBOOK BALANCE'
5600 PRINT 'SHOULD NOW SHOW A BALANCE OF $'; BF
5700 PRINT 'YOU SHOULD NOW DEDUCT $'; T; 'FROM'
5800 PRINT 'YOUR CHECKBOOK FOR CURRENT BANK'
5900 PRINT 'SERVICE CHARGES. SEE YOU NEXT MONTH!'
6000 PRINT
6100 PRINT
6200 END
9000 PRINT '(ENTER AMOUNT OF 0 WHEN THROUGH.)'
9100 LET N=1
9200 LET T=0
9300 INPUT X
9400 IF X = 0 THEN RETURN
9500 LET T=T+X
9600 LET N=N+1
9700 GOTO 9300
```

Execution of the high level program would result in the following type of dialogue taking place between the computer user and the computer system. (Underlined segments signify the user's response to queries.)

```
WHAT DOES YOUR BANK SHOW AS
YOUR STATEMENT BALANCE?100.00
PLEASE LIST INDIVIDUAL AMOUNTS
OF ALL OUTSTANDING CHECKS:
(ENTER AMOUNT OF 0 WHEN THROUGH.)
?15.00
?22.95
?11.05
?.75
?10.25
?0
PLEASE LIST ALL DEPOSITS MADE
AFTER STATEMENT DATE:
(ENTER AMOUNT OF 0 WHEN THROUGH.)
?25.00
?50.00
?25.00
?0
PLEASE LIST ANY BANK SERVICE
CHARGES SHOWN ON BANK STATEMENT
WHICH HAVE NOT YET BEEN DEDUCTED
FROM YOUR CHECKBOOK:
(ENTER AMOUNT OF 0 WHEN THROUGH.)
?6.50
?.50
?3.00
?0
YOU LISTED 5.0 OUTSTANDING CHECKS
WHICH TOTALED $60.0, AND REPORTED
3.0 DEPOSITS AMOUNTING TO $100.0
AS NOT BEING CREDITED ON THE BANKS
STATEMENT. YOUR CHECKBOOK BALANCE
SHOULD NOW SHOW A BALANCE OF $150.0
YOU SHOULD NOW DEDUCT $10.0 FROM
YOUR CHECKBOOK FOR CURRENT BANK
SERVICE CHARGES. SEE YOU NEXT MONTH!
```

The power of a computer really appears impressive when one uses it to quickly obtain information such as, for instance, how results might vary if a parameter in an equation is altered. Engineers, scientists, businessmen, and private citizens frequently need to (or should!) be able to analyze problems and seek solutions through such studies. The computer, with its untiring ability to make iterations, gives people this capability. The following program will serve to demonstrate, as a close to this chapter, how a high level language such as BASIC may be used to program a computer to repeatedly solve a problem while varying the value of a parameter.

Problem: Develop a program that will solve the formula:

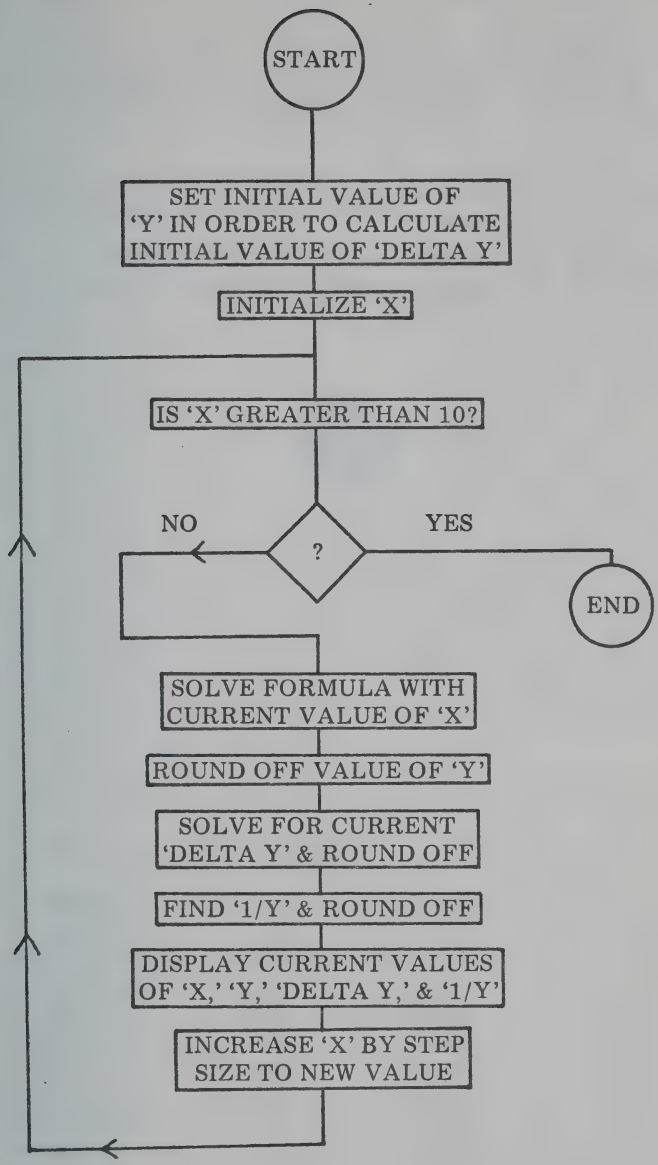
$$Y = X^3 - X^2 + 2X - 4$$

When X is varied (say, in integer steps from 1 to 10). Also, calculate the rate of change of Y as X is varied and find the reciprocal of Y. Round off answers and present the data in table form so that it may be readily viewed and studied.

Solution: Utilize a FOR. . . .NEXT loop with a STEP value of 1.0 to repeatedly solve the indicated formula, obtain the reciprocal value, and rate of change. (To find the rate of change, subtract the value of the N'th solution from the previous (N'th - 1) solution!) This will require a special calculation at the beginning of the program to find the initial rate of change. Use a rounding formula to reduce number of significant digits after each calculation. Organize output into table format.

A flow chart of the proposed program is illustrated. The BASIC language program and its output is shown following the flow chart. Note that by altering just one line in the program (line number 200), the range of values over which the formula was solved and the STEP size (distance between two values of X) could be altered.

Finally, the reader should fully realize that the formula used in the example is irrelevant. The formula used might represent the progress of a chemical reaction when a given amount of catalyst (X) was applied. 'Y' could represent the amount of end product (and 'Delta Y' an indication of when things might start to explode - an important


START
SET INITIAL VALUE OF 'Y' IN ORDER TO CALCULATE INITIAL VALUE OF 'DELTA Y'
INITIALIZE 'X'
IS 'X' GREATER THAN 10?
NO
?
YES
END
SOLVE FORMULA WITH CURRENT VALUE OF 'X'
ROUND OFF VALUE OF 'Y'
SOLVE FOR CURRENT 'DELTA Y' & ROUND OFF
FIND '1/Y' & ROUND OFF
DISPLAY CURRENT VALUES OF 'X,' 'Y,' 'DELTA Y,' & '1/Y'
INCREASE 'X' BY STEP SIZE TO NEW VALUE

```
100 PRINT
120 PRINT
140 PRINT "  X", "  Y", "DELTA Y", "  1/Y"
160 PRINT
180 LET Y1 = -4
200 FOR X = 1 TO 10 STEP 1.0
202 X1=X*100+0.5
204 X1=INT(X1)
206 X1=X1/100
210 LET Y2=X↑3-X↑2+2*X-4
212 Y2=Y2*100+0.5
214 Y2=INT(Y2)
216 Y2=Y2/100
220 LET DY=Y2-Y1
222 DY=DY+100+0.5
224 DY=INT(DY)
226 DY=DY/100
230 LET Y1=Y2
232 YY=1/Y1
234 YY=YY*10000+0.5
236 YY=INT(YY)
238 YY=YY/10000
240 PRINT X,Y1,DY,YY
250 NEXT X
300 END
```

X	Y	DELTA Y	1/Y
1.0	- 2.0	2.0	- 0.5000
2.0	4.0	6.0	0.2500
3.0	20.0	16.0	0.5000E-01
4.0	52.0	32.0	0.1920E-01
5.0	106.0	54.0	0.9400E-02
6.0	188.0	82.0	0.5300E-02
7.0	304.0	116.0	0.3300E-02
8.0	460.0	156.0	0.2200E-02
9.0	662.0	202.0	0.1500E-02
10.0	916.0	254.0	0.1100E-02

consideration if you are the chemist who will monitor the reaction!) No, the formula in the example is not what is important. The concept that is important is that virtually any type of problem that can be similarly formulated can be just as readily solved - whether it applies to mathematical, scientific, business, or problems of personal interest!

The type of program illustrated, once one has a little programming experience in BASIC language, can be developed in a matter of minutes - and executed by the computer in a matter of seconds. Truly, this kind of capability offers tremendous potential for the enlightened!

INPUT/OUTPUT DEVICES

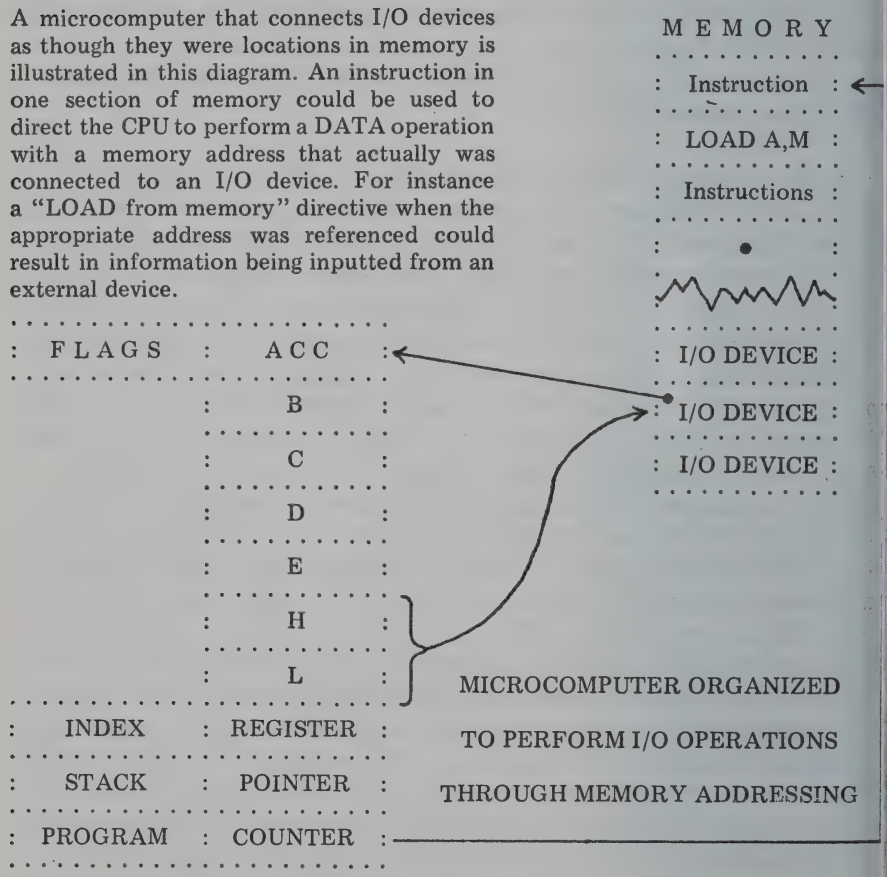
It has been said (and few people who have given it significant thought appear to want to argue the premise) that a computer without facilities to accept information from external devices and transmit information to external output devices is of virtually no practical value. (For any "Engineer" type readers, it is acknowledged that a computer without any input, but with output capabilities, can serve a practical function as a "controller." That being the case, for the purpose of this discussion, let such a machine be called a "controller!") Indeed, a prime reason for the extreme usefulness of the computer is because it is a machine that can accept information, "massage" it (by performing calculations, sorting, restructuring, or whatever) at speeds many times in excess of what the human mind can achieve, and then provide "results" to people by relaying the appropriate information to an output device.

Devices used to INPUT information to a computer or OUTPUT information from a computer are often collectively referred to as "I/O" devices (the result of abbreviating the words Input/Output). For the sake of organizing this discussion, such devices might be categorized into three groups: 1.) Types of equipment used to translate information from human-oriented to machine-oriented form. Such devices would include, for example, electronic typing keyboards (equivalent to a typewriter keyboard) that would convert the action of pressing a selected key (corresponding to a letter or number) into a binary code (such as the ASCII code discussed in a previous chapter), or types of equipment that display information by translating groups of binary bits into human-recognizable letters of the alphabet on a display device such as a printer, video, or cathode-ray-tube. 2.) Types of equipment designed to serve as "mass" or "bulk" memory storage units. These devices can effectively make "copies" of the contents of the memory (or selected sections of memory) of a microcomputer and save them for long term storage on external media (such as paper tape, magnetic tape, magnetic disks, and so forth as will be discussed further in this chapter). When desired, the original contents of a computer's memory (or selected portions) may be restored to the exact duplicate status as that when the "copy" was made. In this manner, entire programs or sets of data may be restored to the computer sc

as to allow continued execution of a program, or further manipulation of a set of data. 3.) "Special" equipment not falling into the first two categories that might be connected to a computer to allow it to, say, take readings from a scientific instrument, ascertain the status of electronic switches, turn on lights or motors, and generally "control" external equipment.

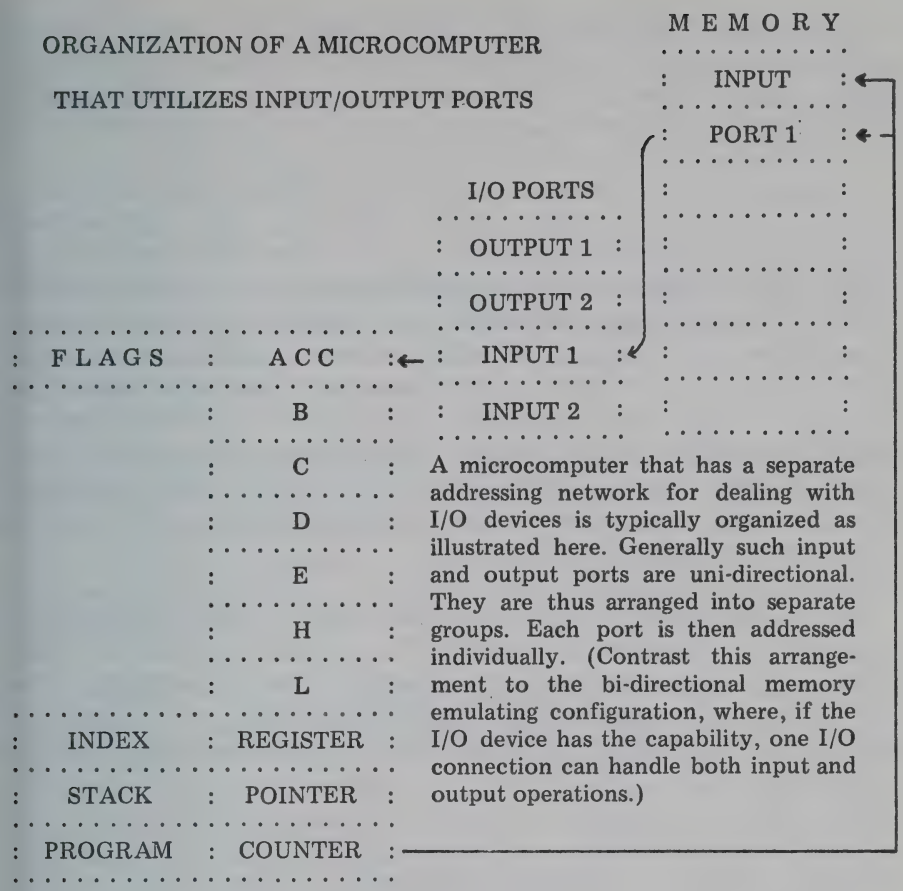
This chapter will introduce the reader to some of the most common types of I/O devices likely to be encountered in a microcomputer system, and provide examples from each of the three categories just mentioned.

A microcomputer that connects I/O devices as though they were locations in memory is illustrated in this diagram. An instruction in one section of memory could be used to direct the CPU to perform a DATA operation with a memory address that actually was connected to an I/O device. For instance a "LOAD from memory" directive when the appropriate address was referenced could result in information being inputted from an external device.



ORGANIZATION OF A MICROCOMPUTER

THAT UTILIZES INPUT/OUTPUT PORTS



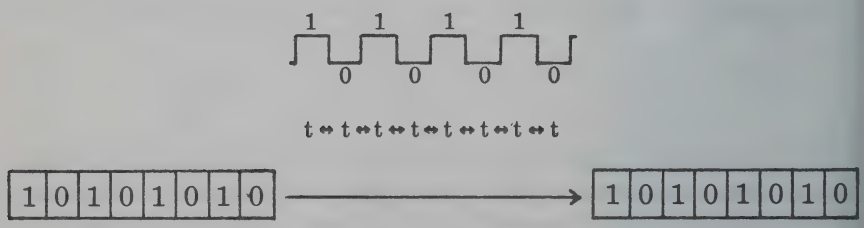
A microcomputer that has a separate addressing network for dealing with I/O devices is typically organized as illustrated here. Generally such input and output ports are uni-directional. They are thus arranged into separate groups. Each port is then addressed individually. (Contrast this arrangement to the bi-directional memory emulating configuration, where, if the I/O device has the capability, one I/O connection can handle both input and output operations.)

I/O operations (that is, the inputting and outputting of information to/from a microcomputer) generally take place either by passing the contents of the primary accumulator to/from so-called "I/O PORTS," or by connecting the I/O devices in such a fashion that they effectively appear as locations in memory, and are referred to as though one were addressing specific bytes in memory. (An I/O PORT may be viewed conceptually as a special "address" outside the bounds of regular memory which, when specified, effectively "addresses" a particular set of connections which are understood to pass to or from an external device and the computer. Conceptually,

one may view the passing of a binary pattern to/from the primary accumulator to/from an I/O PORT just as one would view a binary pattern being "stored" in a location in memory from a CPU register, or "loaded" from an address in memory to a CPU register.)

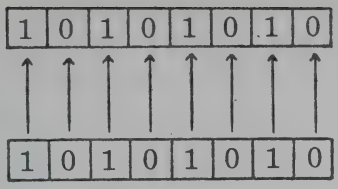
Typically, CPU's that treat I/O devices as though they were locations in memory have no special instructions that differentiate between I/O operations and other types of instructions. Whereas, when I/O operations are performed via I/O ports, special INPUT/OUTPUT instructions are provided for the CPU that enable one to specify the type of operation (whether input or output) as well as to indicate a specific PORT out of a number of available ports. The number of such ports may range from a few to several hundred. When the number of possible I/O ports becomes extensive, I/O directives generally require multiple bytes, with the first byte specifying the type of operation, and following byte(s) specifying the port to which the directive refers.

Before beginning to describe various types of I/O devices, it may be beneficial to show readers the two fundamental methods by which information may actually be transferred to/from a computer. One method is known as SERIAL TRANSMISSION, and the other is known as PARALLEL TRANSMISSION. Quite logically, the first



SERIAL TRANSFER
Transfer takes place over a series of time intervals (t) as bits of information are "shifted" out of one register, transmitted along the connecting path and "shifted" into the destination register.

PARALLEL TRANSFER



Transfer takes place simultanously on all eight lines.

method refers to transferring information bit-by-bit from a multi-bit register along a single path. Each bit is transmitted at selected intervals of time until all the bits within a register (or that need to be transferred) have been processed.

The second method refers to transferring information from one multi-bit register to another in one single operation at a given instant in time. (Almost all operations within a CPU and between the CPU and locations in memory are referred to as parallel operations, because the bits within one register are all passed to another register simultaneously.)

To most readers, at first glance, the parallel transfer method probably sounds like the best method to utilize. It certainly is fast. Why fool around with transmitting information bit-by-bit over a period of time, when one can accomplish the transfer in a single neat operation? Indeed, inside a computer, that is the method of general choice. But, there is (as always) payment to be made for this neat and fast manner of operation! Transferring information in parallel requires that there be as many paths or lines between the source and destination points (registers) as there are bits in the registers. The serial transfer scheme requires but a single path!

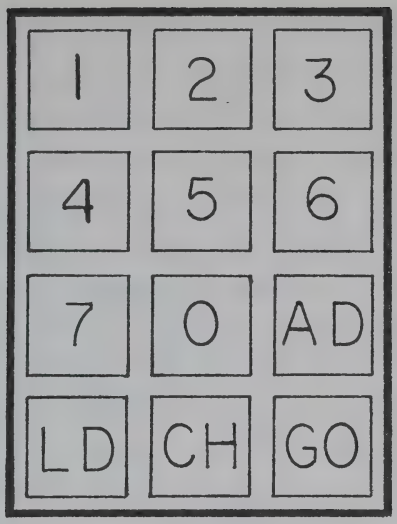
Oh, that single wire feature does not mean very much when one is dealing at the microchip level, when electronic pathways may be thinner than a human hair (by several orders of magnitude). It really starts to count, however, when it comes to interconnecting cabinets full of equipment, such as occurs when one connects a computer to I/O devices. There, wires cost money. Running eight wires to pass information in parallel from a computer I/O port to external devices costs eight times as much as running a single wire and passing the information in serial format. Such wires must typi-

cally be fashioned in cables and joined by connectors, all of which take time to manufacture and require relatively expensive materials. Often, computers are connected to numerous I/O devices simultaneously. Realizing this, the reader may appreciate how reducing the number of such wires can contribute substantually to lowering costs. Hence, the use of the serial transfer technique. Don't forget, though, that while the serial transfer method is physically cheaper, it takes more time to accomplish the transfer. Sometimes, when faced with a choice as to which method to use, one must weigh the cost of transfer time against the cost of materials.

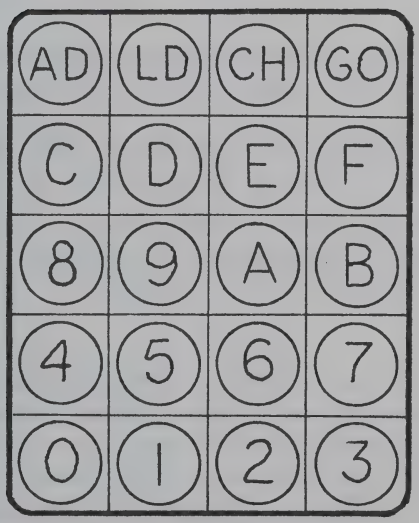
In practice, the trend is towards connecting devices to computers using the parallel transfer technique. This is becoming more and more feasible as the cost of parallel implementation declines due to improved manufacturing techniques, lower cost connectors and materials, and the increased compactness of systems. However, certain types of I/O devices have traditionally used the serial transfer method, and the use of that method will continue far into the future for some classes of devices.

Earlier it was mentioned that one of the major categories into which I/O devices could be grouped was that concerned with converting information from human-recognizable to machine-recognizable form, and vice versa. This category of devices is probably the most important and most frequently encountered. If people could not easily communicate with the computer, it would see little use.

Without question, the most common type of device used to input information to a computer system by human operators is the ubiquitous electronic or electro-mechanical keyboard. In very small systems that require a minimum amount of "personal" input, these keyboards typically contain keys that can transmit binary patterns to represent digits. (Typically the numbers 0 - 7 in order to represent digits used in the octal numbering system, or, the numbers 0 - 9 and digits A - F in order to be able to represent hexadecimal digits.) Additionally, a few extra keys may be provided for special purposes. The accompanying diagram illustrates several styles of these types of keyboards which frequently are arranged in a manner similar to calculator keys.



A COMPUTER INPUT KEYPAD ARRANGED
TO ACCOMODATE THE OCTAL NUMBER SYSTEM



A HEXADECIMAL STYLE COMPUTER INPUT KEYPAD

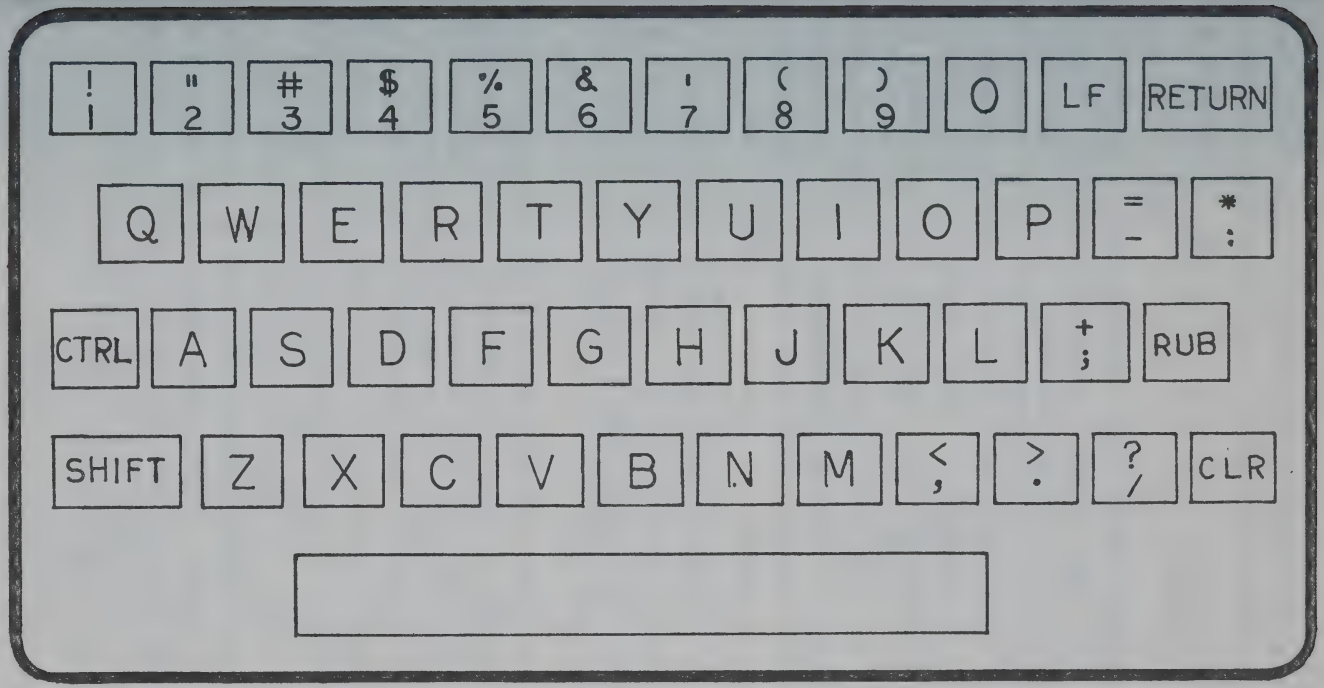
Generally, however, the nomenclature "keyboard" refers to an alphanumeric keyboard arranged in much the same manner as a typewriter keyboard. Keys representing all the letters of the alphabet, the digits 0 - 9, and numerous punctuation marks are contained on the keyboard. When a key on a keyboard is depressed, a binary pattern is generated that may be transmitted to a computer. Most electronic keyboards today generate a binary pattern that can be passed to the computer in parallel fashion. However, some devices, particularly electro-mechanical ones, develop the pattern in a serial fashion. The binary patterns generated may be encoded as the equipment designer sees fit. However, the most popular encoding scheme today for keyboards designed for use in microcomputer systems utilizes the ASCII code described previously in this book. Thus, whenever a particular key is depressed, a selected binary pattern in standardized ASCII code will be transmitted in parallel format along a group of wires that may be connected to a computer.

By typing on such a keyboard, with an appropriate program installed in a computer to accept the data and recognize the meanings of particular characters or groups of characters, a person may readily communicate with a computer just as though one was typing a letter to the machine!

The accompanying picture depicts the appearance of a typical electronic keyboard with full alphanumeric capabilities.

Getting information into the machine is only half the battle as far as the average human operator is concerned. Equally important in most cases (if for no other reason than to verify that what one thought one typed into the machine was actually received) is a device that will display information from the machine in a human-recognizable form.

There are two types of devices commonly associated with microprocessor systems that serve this display purpose. The electromechanical printing mechanism and cathode-ray-tube (CRT) or television (TV) display. The former type has the advantage of being able to produce HARD COPY in the form of permanent imprinting on paper, while having the disadvantages of typically being somewhat noisy, requiring regular maintenance, and somewhat on the expensive



TYPEWRITER-LIKE LAYOUT TYPICALLY FOUND ON ELECTRONIC AND ELECTRO-MECHANICAL KEYBOARDS USED AS COMPUTER INPUT DEVICES

side unless older/surplus units can be acquired. The latter (CRT and TV) displays have the advantages of being relatively inexpensive, silent, and in certain modes, able to display information at a much faster rate than printing mechanisms. It has the disadvantage of not producing a permanent record of the data it displays.

Electro-mechanical printing mechanisms come in a variety of configurations. The most popular styles are somewhat complicated pieces of machinery originally developed for the communications trade and later pressed into service for use with computers. These types of machines are frequently encountered in telegram offices, airline reservation centers, etc. They typically are comprised of motorized mechanical linkages which cause a typing element (or elements depending on the type of printer) to strike an inked ribbon which causes an imprint of a character to be made on paper. The character selected for printing is determined by electro-mechanically decoding a binary pattern. Since these types of machines were originally designed to operate over phone and telegraph lines in the days when cable was at a premium, they are generally designed to accept information in serial bit-by-bit format. Older models of these types of printers typically used a code that is gradually disappearing from use called BAUDOT code, while newer versions operate with the modern ASCII code with which the reader is familiar.

Some modern printing devices utilize new technology to simplify the electro-mechanical design and provide increased reliability and speed with less noise. For instance, there are printers that use tiny thermal devices that actually "burn" special sensistized paper in order to make impressions of characters. Other modern devices use ingenious techniques to greatly simplify the amount of mechanical movement required to print characters. These printing machines may use printing "balls" covered with characters that are spun to the proper location on their spherical surface to print a selected character, special disks with characters mounted on springy finger-like extensions, or fast moving metallic bands or ribbons containing character fonts. Most of these modern printing devices are designed to capitalize on the speed of parallel transfer operations. Many are designed to use the standardized ASCII code.

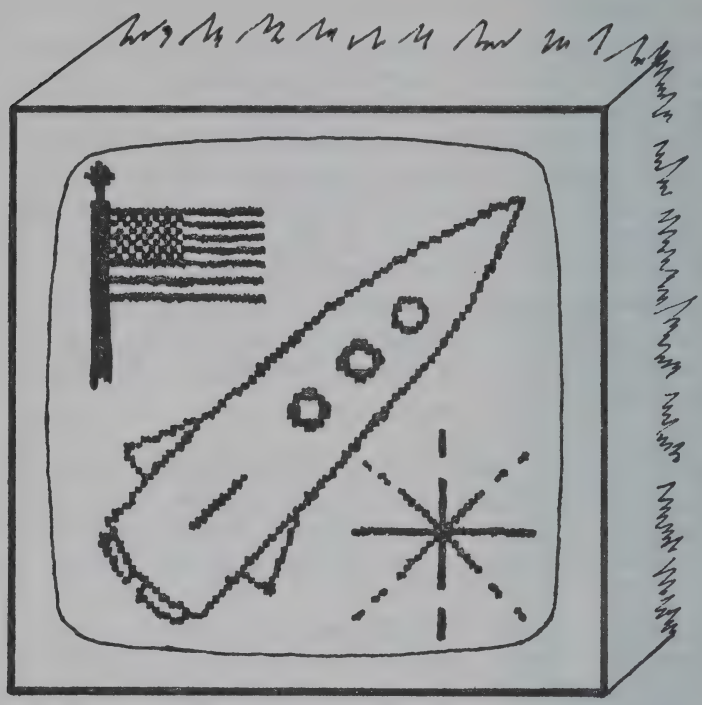
Typical printing devices designed for use with microprocessors

operate in the 500 to 1500 characters per minute range. That is, 100 to 300 words per minute assuming 5 characters per word. Devices operating at these speeds typically print just one character at at time.

At the upper speed range of electro-mechanical printing mechanisms are machines known as "line printers." These are devices that print entire lines of characters at once. Few people can afford to purchase such units for private use. They are more often used by firms that can utilize their speed and absorb their total cost. These types of printers can operate at rates exceeding 10,000 characters per minute, and are truly a sight to watch in operation. Paper filled with text or whatever graphic information is being displayed literally leaps out of such machines when they are in operation. A few of these line printers actually use purely electro-mechanical means to print on the paper. However, others use sophisticated high technology practices such as electrostatically spraying ink on the paper to form characters. While these devices may be out of the average reader's budget range, one can always dream!

Cathode-ray-tube displays permit information to be displayed not only as simple characters, but also in graphic form, such as line drawings or dot-matrices. An increasingly popular version of the CRT display utilizes the cathode-ray-tube found in the everyday television set, and capitalizes on the TV circuitry itself to display information. Using a special INTERFACE between the computer and a television set that converts binary patterns into correct video pulses, a TV set may be used to present not only simple alphanumeric characters, but also graphic patterns. The graphic patterns are produced by effectively forming a grid of points or dots on the screen which may be illuminated singly or in groups to form pictures. The grid of points is formed by continually sweeping the screen horizontally forming lines (while moving down the screen) and applying pulses to the electron beam that causes dots to appear on the screen. The closer together the raster lines are, and the faster the dot-making pulses occur, the more points one may make on the screen and the higher the resolution one will be able to obtain. The ultimate limitations on resolution are dictated by electronic parameters. Conventional state-of-the-art techniques today typically provide grids that may range from 64 X 64 dots on up to 512 X 512 with 128 X 128 to 256 X 256 being a typical size. Graphics are generated by selectively activating desired

points in the grid as the electron raster sweeps the face of the tube. Using this method, one may generate simple pictures, or letters and numbers of various sizes and shapes.

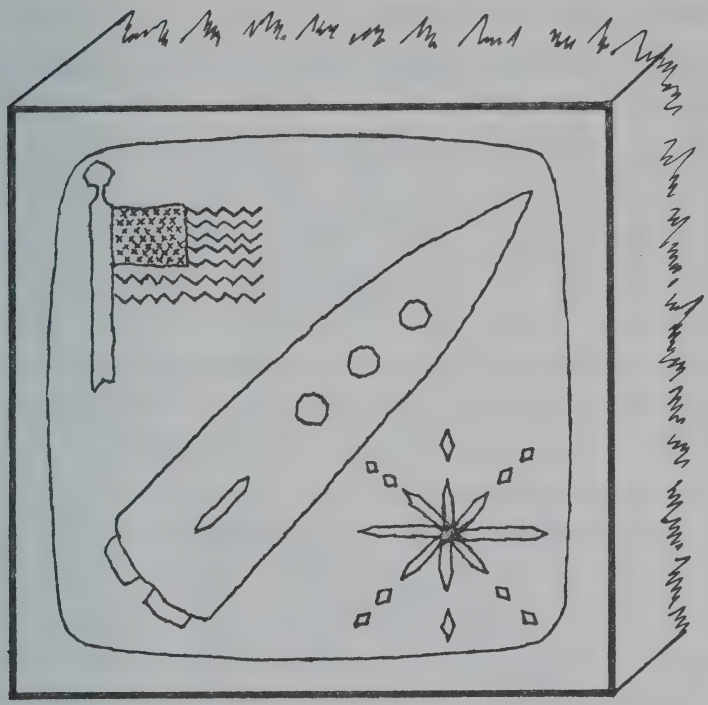


DOT MATRIX CRT DISPLAY

The dot matrix or raster display characteristically has a granular appearance. The smoothness of the display (resolution) is determined by the number of grid points available on the screen which may be individually illuminated or darkened.

A second method of generating pictures and characters on a conventional CRT (such as an oscilloscope display tube) is known as the vector generating technique. Here, instead of steadily sweeping the entire tube face with a raster that is selectively blanked and unblanked, digital signals from the computer are translated to voltages that cause

he electron beam to move from one point to another on the screen. This movement causes lines to appear on the screen which may be used to form pictures or characters of the alphabet. An advantage of



VECTOR GENERATING CRT DISPLAY

A vector generating system draws point-to-point line segments on a CRT screen. The system typically presents smoother looking drawings of objects than does the dot matrix method. Note that the method is well suited for displaying sketches or engineering/architectural type drawings.

this second method over the raster method is that less computer memory is generally required to implement the display. This is because only the starting and finishing points of line segments in a particular display have to be stored (remembered) at any one time. A negative aspect of the method is that a great deal of computer time

must be spent REFRESHING the display on the CRT as the lines have to be constantly redrawn so as to appear in a steady state to the viewer. (The phosphorus in a typical CRT tube must be activated by the electron beam about 60 times a second to avoid a flickering effect.) Failure to keep that beam moving from one point to another will cause the display to "go out."

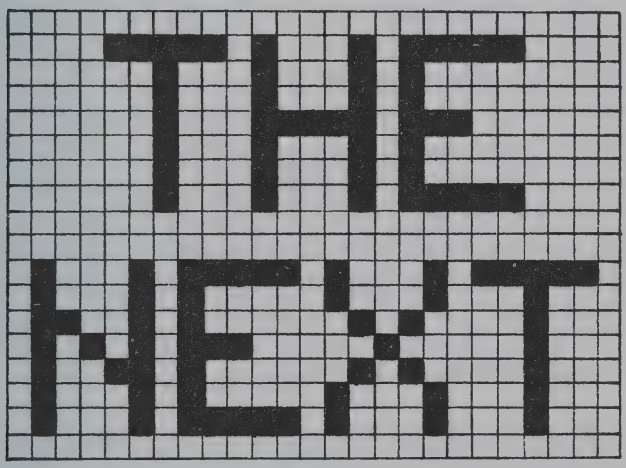
The raster (dot-matrix) display method has its drawbacks too. For one thing, it takes a great deal of memory to store the status of all the points in a dot matrix. The status determines whether a point will be illuminated (unblanked) or extinguished (blanked). Each grid point (dot) requires one bit of memory in the computer to indicate whether it is "off" or "on." A display with moderate resolution having a grid of 256 by 256 points containing some 65,536 dots. That means that a microcomputer having 8 bits per word of memory would need to dedicate 8,364 words of memory for the sole purpose of controlling the display. That is a lot of memory to have to devote to display purposes, especially considering that other types of display devices may only require that a few bytes of memory storage be dedicated to the outputting of data! (Remember, every point in the dot matrix must be represented in the display even if a particular pattern being displayed has 99.9 percent of the points "off.") Not only must this memory be supplied in the system, but quite often it must be especially designed as a separate entity devoted to display purposes only. Or else, it must be equipped with a capability known as DIRECT MEMORY ACCESS (DMA). This special DMA capability allows the memory elements to be accessed (addressed and read) by the display interface when it is not being accessed by the CPU! Naturally, providing this extra capability in a system costs more money.

Does the raster method have worthwhile advantages? Yes - in addition to the fact that it is often designed so that it can use an ordinary TV set as the display tube, it can also be configured to drive a color TV set and display information in color. Additionally, since a special block of memory must be devoted to the display, the computer has only to set up the display pattern whenever there are changes. It does not have to continually spend time REFRESHING the display as occurs when a computer drives a vector display. Thus, the computer

can perform other tasks more efficiently when coupled to a raster display. Finally, the system is somewhat more stable in certain aspects than the vector generating method. This is because the method employs all digital techniques from computer through display process, whereas the vector technique requires conversion from digital data to analog signals. This latter process requires care on the part of the circuit designer and typically requires somewhat more adjustment than the purely digital method.

Both types are commonly encountered in personal computing systems although it does appear that the purely digital approach is somewhat more popular than the vector method. One reason for this popularity is undoubtably due to the fact that many of the interfaces utilizing that method will directly connect to an ordinary TV set which virtually everyone has readily available.

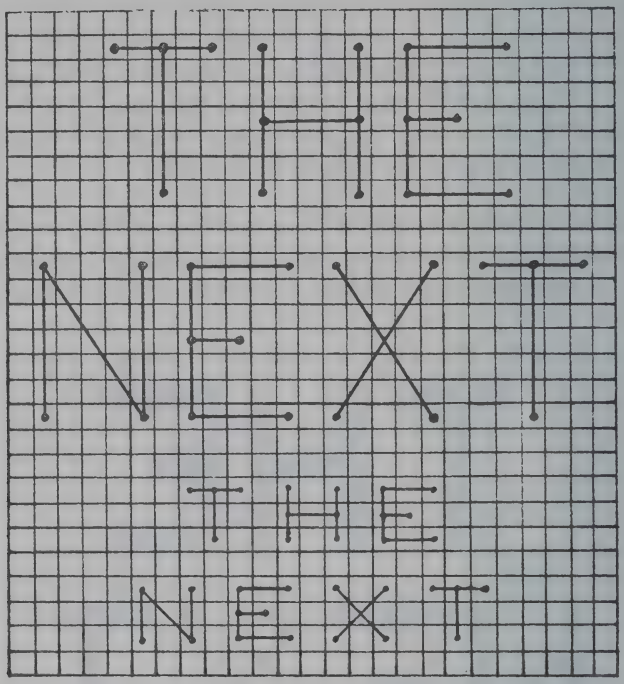
A parameter that one should take into consideration if one is considering purchasing or constructing a CRT display (whether using the digital or digital-to-analog technique) is how many alphabetical or



CHARACTER DISPLAY IN A RASTER SYSTEM

When displaying small alphanumeric characters an all-digital system typically illuminates the appropriate dots in a 5 X 7 matrix. (The "dots" appear as tiny squares or rectangles on most units.)

alphanumeric characters may be displayed on a single line of the display. Individual characters are produced by illuminating selected dots in the all-digital method. Here, for instance, a display having a total screen grid consisting of 256 X 256 points is subdivided into tiny blocks of dots having, for instance, matrices of 5 X 7 dots. Individual dots within each block are illuminated as desired to form the different characters of the alphabet. (See the accompanying diagram of character formation in an all-digital CRT display system.)



CHARACTER DISPLAY IN A VECTOR SYSTEM

The vector method draws lines between points. When superimposed for illustrative purposes upon a grid such as that used in a raster system one may observe that: 1.) Only the ends of line segments need to be identified (specified by the computer) hence less points have to be dealt with by the computer or display interface. 2.) The vector method can often present characters in a smaller area of the screen because the line segments provide finer resolution.

Characters may be formed in a vector generating display by drawing lines from one point on a screen to another in a particular sequence as shown in the appropriate diagram.

Regardless of which type of CRT display is being utilized, there will be a limit to the maximum number of characters that may be displayed on a single line. In some units, the number of characters allowed per line may only be 32 or so characters. Other models may allow 64 characters, still others up to 80 characters on a single line. As a general rule, the more characters that can be displayed on a single line, the more expensive the display system will be! Many CRT displays that rely on the use of a TV set for displaying the information are limited to 32 to 40 characters per line. Thus, one would want to choose carefully if one wanted a system primarily, say, to edit legal documents that normally contained 70 to 80 characters per line. On the other hand, if one planned to just edit newspaper columns that consisted of short (width) columns, one could do well enough with a display limited to a lesser number of characters per line. Another rough rule of thumb for a newcomer to go by is whether one plans on a system that will expect heavy use of a high level language, or whether one plans to develop programs almost exclusively using machine language methods. A guideline here is to go for the maximum number of characters per line one can afford for a high level language oriented system. One can get by with less capability if one is going to be doing heavy work in machine language (because of the nature of assembler language for one thing). Furthermore, the machine language programmer will be in a better position to tailor programs to the system's individual capabilities. The high level language programmer may not have the inclination (and might find it hard to obtain information regarding thereto) or ability to develop or modify machine language programs, such as an interpreter, so that it can put a long formula on two lines of the display!

In addition to the number of characters one can display on a single line, there is also the consideration of how many lines of information may be presented on the CRT screen. For CRT systems, this typically ranges from as few as about 8 lines up to 24 or more. This parameter is important because, combined with the one just discussed, it sets the upper limit on how much information may be displayed to the viewer at one time. Once the CRT screen is filled, some of the

information being displayed will have to be removed from the screen before new data can appear. This is somewhat different from a printing device which can continue to print an almost limitless number of lines of paper that may be read at the operator's convenience. Typically, a CRT display screen is controlled so that new lines of information appear at the bottom of the screen with preceeding lines moving up the screen. When the screen becomes completely filled, the top line of information is removed from the display each time a new line of data appears at the bottom. This process of "rolling" the lines of information upwards as new lines appear, and removing the topmost line, is called SCROLLING. (Fancy CRT display systems sometimes provide capability to be able to roll lines back down in order to review previously displayed data. This process, however, is really accomplished using programming techniques, and is a result of adding special CRT display controlling capabilities of Editor type programs.)

The fact that only a limited amount of information may be displayed at any one time, and the fact that the displayed information is not readily converted to a permanent "hardcopy" form, are the major drawbacks of the CRT system. Its advantages, however, come from the fact that it is a relatively inexpensive method of displaying information compared to electro-mechanical printers, and that it can be designed to present information at considerable speed. Additionally, the vector method provides graphic capabilities not easily duplicated in an electro-mechanical device. (Special X-Y plotting devices which will be mentioned later, while utilizing similar display concepts, are no match for the speed at which a vector CRT operates, and despite their slow speed, are considerably more expensive!)

While the discussion of keyboards and displays has presented the two types of devices separately for introductory purposes, it may be apparent that these two types of devices are frequently provided together as one package forming an operator's input/output console. Using such a console, an operator is able to conveniently communicate with a processor. Frequently, such an I/O console is referred to as a communications TERMINAL. Such terminals frequently pair an electronic keyboard with a CRT display. Or, they pair an electro-mechanical keyboard with an electro-mechanical printing device.

The combination of an electro-mechanical keyboard and an electro-mechanical printer has been around for a long time in the communications industry, and is frequently referred to as a teletypewriter. Modern teletypewriting machines designed for use in computer systems utilize the standard ASCII code patterns previously described for the reader. However, older machines (which are still around in considerable abundance) were designed to utilize a set of binary patterns known as the BAUDOT code.

Baudot code teletypewriting machines are often available at low prices from surplus suppliers, and hence, frequently are obtained by computer system owners interested in using them as low cost I/O terminals. (Any readers who are familiar with amateur radio or the communications industry have probably observed these older "iron monsters" still "clunking" (as they are somewhat noisy) away reliably after many years of steady service.) Since these machines do not utilize the standard ASCII patterns for letters and numbers, it is necessary for those using them with computers to either develop programs to convert between ASCII and BAUDOT code, install electronic circuits between the computer and the machine that will perform that function, or design programs that have all I/O operations tailored to utilize the Baudot code. A computer is easily programmed to perform the conversions necessary between the two codes. Some people, however, prefer to utilize HARDWARE (electronic circuit) conversion techniques. The option to design programs that utilize the Baudot code exclusively works fine, but it tends to put the program into a category that does not lend it to exchange with other computer users. This is because it does not adhere to the standardization principles that are necessary to foster easy communications amongst users.

How different is the Baudot code from ASCII code? Quite a bit. Baudot code was developed as a five-level code, meaning it could be represented by the machine decoding patterns that can be contained in five binary bits. But wait! The reader may calculate that five binary bits can only represent 32 possible patterns. It would seem that a typing machine that was going to be able to print the 26 letters of the alphabet, ten digits, and be able to handle some punctuation marks, certainly needs to be able to process more than 32 different symbols! How can that be done with a machine that operates by decoding just 32 different patterns?

The original designers of Baudot coded machines came up with a pretty clever way of being able to print a lot more than 32 different symbols while still using a five-bit binary code that only permitted 32 patterns! They assigned two of the 32 patterns to tell the machine to shift from lower to upper case, or vice versa. Then they assigned two symbols each to most of the keys/printing elements on the teletypewriter. Bingo! Almost double the number of symbols could be handled by the machine even though the number of binary bits used to pass information was not increased. When a machine is in one mode of operation, say, upper case, because it has received the binary pattern meaning "shift to upper case," it prints numbers and punctuation according to the five-bit binary patterns it receives. When in the lower case mode after receiving a "shift to lower case operation," it prints letters of the alphabet. A few special operations, such as carriage return, or the spacing operation (where no character is printed) are interpreted the same regardless of which mode the machine is in. This was done because those characters occur so frequently that assigning them to just one mode would result in too much "shifting" between modes.

The individual who wants to use a Baudot coded device with a computer, however, has to be careful. This is because one may be lulled into a false assumption. Remember, these Baudot coded machines were originally developed to be communicating with one another. Thus, for example, when one prepared to send a message from one machine to its counterpart, the operator would start a message by striking the appropriate key to indicate in which mode the machine was to operate. Once received, this information was remembered by the machine (mechanically) until it received the pattern instructing it to shift to the opposite mode. Note that in fact, the machine itself served to remember one bit of information - the mode that it was operating in! Only five bits of information are transmitted from one terminal to another. However, in order to represent all the possible characters, it actually requires six binary bits of information! The fooler here is that the machine serves as a logic indicator holding one bit of information! Therefore, when a computer is connected to such a device, it must mimic the ability of the machine to remember which mode it is operating in (upper or lower case), and this can only be done by adding another bit to the standard five bit Baudot code! The accompanying table illustrates binary patterns for

the standard five-level Baudot code as it would appear when utilized by Baudot coded teletypewriting machines themselves. It also illustrates how these patterns would appear in the memory of a microcomputer (using octal notation) realizing that a sixth bit of informa-

BAUDOT CHARACTER SET

CHARACTERS LC	CHARACTERS UC	5 LEVEL CODE BIT POSITION 5 4 3 2 1	OCTAL CODES LC	OCTAL CODES UC
A	-	0 0 0 1 1	003	043
B	?	1 1 0 0 1	031	071
C	:	0 1 1 1 0	016	056
D	$	0 1 0 0 1	011	051
E	3	0 0 0 0 1	001	041
F	!	0 1 1 0 1	015	055
G	&	1 1 0 1 0	032	072
H	#	1 0 1 0 0	024	064
I	8	0 0 1 1 0	006	046
J	'	0 1 0 1 1	013	053
K	(	0 1 1 1 1	017	057
L	)	1 0 0 1 0	022	062
M	.	1 1 1 0 0	034	074
N	,	0 1 1 0 0	014	054
O	9	1 1 0 0 0	030	070
P	0	1 0 1 1 0	026	066
Q	1	1 0 1 1 1	027	067
R	4	0 1 0 1 0	012	052
S	BELL	0 0 1 0 1	005	045
T	5	1 0 0 0 0	020	060
U	7	0 0 1 1 1	007	047
V	;	1 1 1 1 0	036	076
W	2	1 0 0 1 1	023	063
X	/	1 1 1 0 1	035	075
Y	6	1 0 1 0 1	025	065
Z	"	1 0 0 0 1	021	061
SPACE		0 0 1 0 0	004	004
CAR. RET.		0 1 0 0 0	010	010
LINE FEED		0 0 0 1 0	002	002
NULL		0 0 0 0 0	000	000
FIGURES		1 1 0 1 1	033	033
LETTERS		1 1 1 1 1	037	037

tion would need to be stored with each pattern. The sixth bit would have to indicate whether the machine should be operating in the upper or lower case mode when the normal five-bit pattern was transmitted. The table shown uses the convention that a logic one for the sixth bit indicates upper case mode. (Sixth bit position shown as the bit to the left of the five bits making up the Baudot pattern.)

Teletypewriting machines consisting of a combination of a keyboard on which one may pass information to a computer, and a printer from which one may receive data are often additionally equipped with machinery that can punch patterns on strips of paper tape, and perform the reverse function (read strips of paper tape containing punched holes and transmit the patterns). This kind of capability is of special importance to computer system operators

Why is such capability important to computer users? Because it provides a means of making copies of the contents of a computer's memory banks! Why is this valuable? Because any computer system is limited to a finite amount of memory that can be addressed and accessed or manipulated within the computer. Typically, microcomputers today can be equipped to have "on-line" anywhere from a few hundred bytes of memory on up to, in most cases, a maximum of 65,536 bytes of memory. Memory elements making up the main memory banks of a computer are expensive too - at least when compared to strips of paper!

The main memory within a computer is primarily made up of electronic elements having what is known as "read and write" capability. This simply means that the CPU is able to read instructions and data from locations in memory as well as place data (or instructions) by "writing" (altering) the binary patterns at locations (addresses) in the memory banks. Most memory elements used in microprocessor systems today are made up of semiconductor materials, and these elements are arranged in groups or blocks of consecutive memory addresses. A CPU in a microcomputer is able to selectively address any desired location in memory due to the way in which the memory elements are configured. A computer arranged such that any arbitrary address in memory may be accessed (regardless of which location was last addressed) is commonly referred to as being equipped with Random Access Memory, which is abbreviated as RAM memory. How-

ever, and this is important to the serious reader, the use of the terminology RAM memory today carries several connotations that cannot be gleaned from the disclosure of the abbreviation's origins.

The connotations carried by the term RAM as applied to small computer's memory elements generally include the following: That the memory elements are derived from a particular electronic technology having to do with materials known as semi-conductors. Consequently, because of the technology, these memory elements do not retain patterns placed in them when electric power is removed. And, when electric power is first applied, these elements will contain completely random patterns of binary bits. Additionally, the terminology typically implies that the devices are "read and write" in that their contents may be readily modified while operating in the computer as a normal course of computer operations. Knowing these connotations associated with the abbreviation RAM is important because there are types of memory using different technologies that have different characteristics. These will be discussed in a little more detail in a successive chapter.

Since there is a finite limit to the amount of memory that can be on-line in a computer, there is a corresponding limit to the size of a program, or number of different programs, and/or the amount of data that may be stored in the memory banks of a computer at any one time. What happens when all of the possible locations in a computer's memory have been assigned for various purposes and one has, say, a large amount of data that needs to be examined? (Possibly, for instance, in order to calculate an average value.) Or, what can one do if one spends a few hours or days developing a computer program and has it stored in the memory banks of a computer, and then needs to utilize the computer for some completely different purpose (which, it may be assumed, will require the use of the memory locations assigned to the program one has been working on)?

What can one do? One can make a VIRTUAL COPY of the contents of memory locations on some type of external device, while maintaining a correspondence with memory addresses (so that the information may eventually be loaded into the same locations)! One way to duplicate the binary patterns stored in memory locations on a

strip of paper would be to punch holes in columns across a row whenever one wanted to represent a binary one value. The absence of a hole in a column would represent a logic zero.

If the columns on a strip of paper represent the bit positions within a word of memory, a row consisting of eight columns could be used to represent the pattern within a word in memory. Sequential rows on a strip of paper could represent consecutive locations within the memory of a computer. By outputting the contents of memory locations, starting at a specific address, to a device that could punch holes in a strip of paper, one could make a duplicate of the contents of memory in a microcomputer. Later, if one desired to "initialize" the contents of memory back to that exact same status, one could "read" the holes in the paper strip (using a PAPER TAPE READER) and transfer the patterns on the paper strip back into memory locations!

Presto! One now has the concept by which the power of the microcomputer can be vastly increased. The reader now knows how a computer can become such a versatile instrument. It is because one may, with suitable I/O equipment, make a copy of the contents of the computer's electronic memory - whether the contents be sequences of instructions that direct the operation of the machine itself, or whether those patterns represent data to be manipulated by the machine, and save that copy for as long as desired in an essentially inert form. Then, whenever one desires, the copy may be placed back into the computer's electronic memory to restore the computer to exactly the state (meaning capability!) that it was in when the copy was made! Equipment used to make copies of the patterns in a computer's memory is often referred to as BULK STORAGE equipment. The medium on which the copy resides while it is outside the electronic memory of the computer is commonly called the BULK STORAGE MEDIUM.

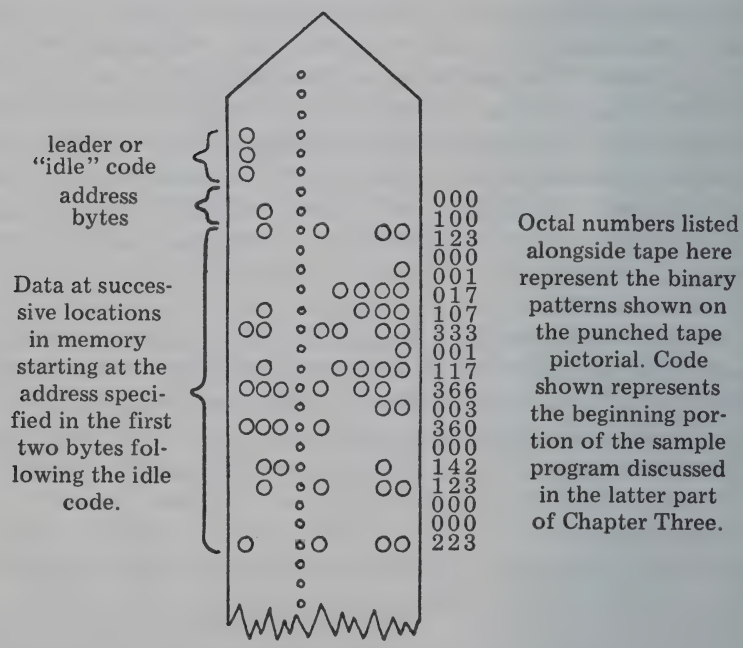
The reader should stop for a few minutes to ponder the importance of what has just been discussed. It has a truly profound effect on the overall usefulness of a computer system! Just think: one minute the computer may be "loaded" with binary patterns that represent instructions guiding it to solve a sophisticated mathematical problem.

Those instructions may be arranged to handle the problem as a "general case" solution in that, given any set of parameters, it can solve for unknown values. Thus, the program that the series of instructions represents has long term use potential even though one may only occasionally have that type of problem to solve. The program may be used to solve one such problem which might be at hand at the moment. The sequence of instructions comprising the program (and even the data used in a particular problem if desired) may then be "saved" on an external bulk storage device until the same problem solving capability is needed at some future data. In its place, the same locations in the computer's memory, using an appropriate bulk storage I/O device to load in new patterns, may be immediately filled with a new sequence of instructions. These new instructions could direct the computer to perform an entirely different type of task. The new type of task could be as diverse as playing a game of checkers with a person, or controlling equipment such as a metal lathe or milling machine. Thus, in a matter of a few moments, the computer can be "transformed" so that it performs a completely different type of task. Not only can external bulk storage devices be used to quickly change the program in a computer, but, as may be readily apparent, they can be used to store almost endless (there are practical limits for various types of storage media!) amounts of data. It is through the use of external mass storage devices that computers are able to process vast lists of names and addresses, say, in order to create selective mailing lists; or to create updated weather maps by correlating the measurements obtained from hundreds of weather stations.

A teletypewriting machine equipped with a PAPER TAPE PUNCH and a PAPER TAPE READER may be considered a BULK STORAGE or MASS STORAGE I/O device. If the machinery is designed to utilize the standard ASCII code, the paper tape is nominally one inch in width. Provisions for punching binary information in eight columns across the width of the tape are provided - seven columns for representing the full ASCII code, and an eighth column for optional use in error checking schemes, or for other purposes. (Note how convenient this format is with microprocessors holding eight bits in a word of memory!) In addition to the eight "information carrying" columns provided for on the tape, a special column near the center of the tape strip is provided for sprocket feed holes. These

holes facilitate mechanical movement of the paper tape when it is being processed by the tape reading portion of the equipment. The sprocket feed holes are noticeably smaller than the holes used to represent binary bits.

Baudot encoded equipment uses a paper tape that is typically about 11/16 inch wide. It too has a sprocket feed column near the middle of the tape that is punched with small holes that are about half the size of holes formed to represent binary information. Since the Baudot code only utilizes five binary bits, only five columns of information holes are provided for across the tape. This makes the paper tape equipment associated with Baudot machines somewhat less efficient when used with an eight bit microprocessor, as a row of columns on the five column wide paper tape cannot represent all the bits in a byte of the microprocessor's memory. However, such equipment is sometimes used as a bulk storage device by simply using two rows on the tape to represent the data in one byte of memory.



PICTORIAL OF PUNCHED PAPER TAPE

The accompanying pictorial illustrates how binary information residing in bytes in a computer's memory may be transferred to a punched paper tape. The pictorial assumes ASCII encoded equipment is being utilized. The pictorial also illustrates how a simple, commonly used tape formatting technique may be used to automate the process of loading information on the tape back into the correct memory addresses in the computer. This technique consists of placing the binary value of the address in memory where information stored on the paper tape is to commence being loaded, as the first two rows on the paper tape! With that information available on a tape, a computer program to read in the paper tape from the paper tape reading equipment may be designed so that the loading process is automatic. The computer reads the first two rows on the tape, thereby learning the first address in memory where the next row on the paper tape is to be stored. It then continues to read in rows from the tape, placing the binary patterns in successive locations in memory, until the tape runs out. (Or, until, say, a special coded pattern on the paper tape informs the program to stop reading the tape, or change the loading address to a new location, etc.)

There are several factors that determine the desirability of bulk storage devices for use with microcomputers. Unfortunately, some of the most important ones conflict in that it is difficult to maximize one without sacrificing performance of another. For instance, the speed at which a bulk storage can transfer data from/to the computer's memory to/from the bulk storage medium is a parameter which one would typically like to see maximized. The faster at which such exchanges from the stored mode (outside the computer) to the active mode (inside the electronic memory elements of the computer) can be achieved, the more efficient one may consider the system. After all, not much is being accomplished during the time a computer has to wait for new data to be read in from a bulk storage device. If a mass memory device can transfer new data at a rate of 1000 bytes per second, it will take a lot less time to execute a transfer than if the rate is 10 bytes per second. Unfortunately, as a rule of thumb in the past, the rate of speed at which a mass memory device could access and transfer data correlated to its cost. The more it cost, the faster it operated. While modern technology is starting to put some kinks in that correlation, it still is a serious consideration in practical systems today.

Another parameter that is important to consider is the reliability of the mass storage device and medium. Once again, as a general rule, increasing reliability increases costs. Notice that these costs must be considered not only from the viewpoint of the bulk storage equipment, but from the standpoint of the storage media cost (on a "cost-per-byte" basis).

Adding paper tape punching/reading capability to a typewriting machine as an option generally runs a few hundred dollars (on a new unit). Such paper tape handling equipment can typically process paper tape at a rate of from 10 to 30 characters per second, which is quite slow by today's standards. (Loading a high level language program into the memory of a computer from paper tape at this rate can take, for instance, 6000÷30 to 6000÷10 seconds (200 to 600 seconds), or from a little over three to as much as ten minutes if the tape was efficiently formatted. Many tape formats require double to triple this amount of time.) A roll of paper tape containing about 1000 feet or enough for recording 12,000 bytes costs about $3.00. Cheap enough one might say for a recording media until one compares the cost-per-byte figure against that for some other media that will be discussed in this chapter. Paper tape as a recording medium is highly reliable under normal use. It is repairable if slightly damaged by tearing or creasing, for instance. One can even "recover" information from the tape in dire circumstances because the patterns (holes) may be discerned by the human eye quite readily. Put all the above parameters together. Is the bulk storage system using paper tape economical? To some extent it depends on the individual's circumstances (can used equipment be obtained?) as well as the individual's preferences. (Does one prefer the type of reliability that comes with using a medium that can be visually verified?)

The types of paper tape punches and readers that can be coupled to teletypewriting machines are generally restricted to rather low speeds of operation. This is really due to the fact that when such devices are coupled to the teletypewriting machinery they become limited by the ultimate speed of those machines. One can obtain specially designed paper tape readers and punches that operate at speeds much higher than 30 characters per second. Paper tape reading devices are available that will readily process paper tapes at 200 to 300 characters per second. Specially designed units can read punched

paper tape in excess of 1000 characters (rows) per second. Of course, as a general rule, the higher the speed at which the device can operate, the more expensive it is in price. These high speed readers are generally motorized so that the paper tape is automatically fed through the reader hole-detecting mechanism. Quite often the types of motors used are known as STEPPING MOTORS because their shafts may be rotated in precise steps of a fixed number of degrees. The use of stepping motors allows the paper tape to be read in a one-row-at-a-time manner. Under computer control, the tape is incrementally fed through the hole-sensing portion of the reader. This precise control offers advantages when processing certain types of information because it allows the computer to control exactly when another character is to be read by the mechanism and gives full control of the process to the computer. However, stepping motors are expensive and contribute substantually to the cost of a high speed paper tape reader. Motor driven paper tape readers typically retail new for between $300.00 to $1000.00, depending on operating speed and other technical parameters. Recently, however, extremely simple paper tape readers that do not use motors, but rather allow a person to simply pull a paper tape through a hole-sensing mechanism, have been introduced. These types of devices are quite low cost (under $100.00), and can operate at surprisingly high speeds (reported at from hundreds to well over a thousand characters per second). However, since the readers are not under the precise character-by-character control of the computer into which information is to be read, their versatility is somewhat diminished. While they appear suitable for the simple loading of programs into memory, they are not ideal for operations such as reading in "blocks" of data which would require frequent starting and stopping of the tape.

High speed paper tape punching machines are also available. Punching holes in paper tape is a somewhat different process than determining whether holes are present. As might be imagined, these devices are somewhat more expensive than high speed readers. They are also considerably slower because of the physical hole punching processes involved. Typically, so-called "high speed" paper tape punches are designed to be able to punch from 70 to 200 rows per second. They sell in the $1000.00 to $3000.00 range. Fortunately, most individual microcomputer system owners can get by without having a high speed paper tape punch at their facility. Such punches

are generally used by organizations that produce programs for distribution, and find the increased speed at which such tapes can be produced by the special equipment to be economically advantageous.

Often, a particular manufacturer will offer a product which is marketed as a combination high speed reader and punch. Actually, such pieces of equipment consist of the two individual units placed together in a package. The package may reduce costs over the individual items slightly, due to the sharing of interfacing electronics to the computer. Again, such pieces of equipment, unless purchased surplus or second hand, command prices that generally limit them to use by well-endowed organizations.

Paper tape, while still being widely used, particularly among those that have the associated reading and punching equipment as part of teletypewriting equipment, is finding declining use among microcomputer users (with a few exceptions for special applications). This is because the medium is somewhat bulky and not as easy to handle as alternative mediums available today. Also, the method is painfully slow in many applications. Furthermore, the medium is not suitable to computerized applications involving the searching and sorting of data and so forth. This is because the ACCESS TIME - the time required to find a particular FILE or block of data - is comparatively slow. Such accessing frequently requires manual operations (such as locating the desired tape or segment on a tape).

An exception to the above may be found amongst microcomputer users who utilize the small computer around heavy machinery, such as in the case for computerized process controllers. In such applications, the computer system may be in a hostile environment containing dirt and metallic or abrasive dust. Also, the recording medium may come in close proximity to pieces of metal and steel or electric motors that may be magnetized. In such instances, paper tape is held in high esteem over alternative methods that rely on magnetic mediums. The magnetic medium systems cannot tolerate the type of harsh environment mentioned. This is especially true in regards to the medium coming in contact with magnetized materials that could alter the data stored as magnetic patterns on magnetic tapes and discs such as those used in the types of bulk storage devices to be discussed next!

Undoubtably, the current most popular bulk memory device in use by microcomputer system owners is the magnetic tape storage unit. These types of units utilize magnetic tapes such as those used in magnetic audio tape recorders in many instances, or tapes which look the same but are actually manufactured to higher quality standards than typically required by the audio recording industry. The tapes for these units are sometimes in the tape CARTRIDGE form familiar to audiophiles, or appear similar to those used in stereo audio tape units. The more popular, however, are the smaller CASSETTES. (A few people even use old-fashioned open reel-to-reel magnetic tape units.)

Reasons for the current popularity of the magnetic tape bulk storage method include the following: Relative low cost of the equipment and medium used. Ease of handling and storage. Increased speed of data transfer to/from the computer - typically 10 to 25 times (and in premium units, even faster) higher than teletypewriter-connected paper tape systems. (Though some magnetic tape systems do not operate much faster than 30 bytes-per-second, which is only three times faster than most teletypewriter-driven paper tape devices.)

To take each asset one at a time, consider the following. Magnetic tape handling equipment that can be used to both read and record information carries a price tag in the range of $25.00 (for a low cost audio tape recorder) to $400 - $500 (for excellent quality digital tape decks). Performance is more or less pretty much related to costs, so the user may select according to needs and budget. This is a price range that is in the vicinity of what very slow paper tape handling equipment runs when coupled to a teletypewriter, and considerably less than what automated high speed paper tape equipment typically costs. Magnetic tape offers increased performance at equivalent prices.

The medium itself - the magnetic tape - costs less per byte of information stored than paper tape. A cassette of magnetic tape costing $3.00 or less can typically hold several hundred thousand bytes of information. $3.00 of paper tape as previously indicated will only provide for recording about 12,000 bytes! But, here is the kicker - magnetic tape can be effectively erased and used over and over again when information becomes outdated! Holes are not so easily removed from paper tape! For all practical purposes, when what is stored on paper

tape is outdated, the tape must be thrown away.

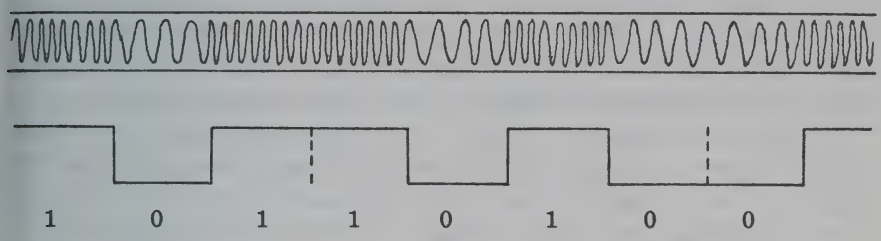
A roll of paper tape is somewhat unwieldy in terms of handling and storage. The problem becomes more apparent when strips of paper tape, say 50 to 100 feet long, containing programs or data have to be physically stored - hopefully in a manner that will allow them to be retrieved when the information they hold needs to be recalled! It doesn't take long when using a paper tape system before one has boxes, file drawers, and various nooks and crannies around the computer installation filled with varying sized rolls of paper tape. Contrast that to a cassette or cartridge containing magnetic tape all neatly reeled inside! Placing a gummed label on the cassette will allow one to list the contents of the tape (in the order in which they occur if a tape contains multiple entries). Furthermore, a complete cassette or cartridge may be stored in a case or tape carrier for safekeeping. If worse comes to worse, cassettes stored in a desk drawer take up a lot less space than rolls of paper tape.

A magnetic tape unit with an electronic interface designed to allow transfer of data from the computer to the tape (record) and vice versa (playback) typically operates at rates which may start as low as 30 characters-per-second, but more generally ranges from 100 to 300, and in special types of units, 600, 800, and more characters-per-second.

(Units above the 600 character-per-second mark generally have price tags that put them out of the range of the individual small computer user's budget. These very fast magnetic tape units typically utilize wide strips of magnetic tape containing multiple recording tracks so that, for instance, eight bits of information may be recorded in one parallel operation. The types of tape units being discussed here for typical use in microcomputer systems generally record data on just one track at a time on a bit-by-bit basis.)

The most popular type of magnetic tape system used by personal computer owners is based on the adaptation of ordinary audio tape recorders. Using this method, a special interface between the computer and the tape recorder converts binary bits (logic levels) into audio tones. One popular convention represents a logic one level as a tone of 2400 cycles per second, and a logic zero level as a tone of

1200 cycles per second. As bits are received serially from the computer, the corresponding tone is fed to the tape recorder for a set amount of time. The accompanying diagram depicts binary data being converted to two different tones for recording by a magnetic tape unit.



Two tones (frequencies) recorded on magnetic tape - such as the kind used for audio recording - may be used to represent the binary logic values of '0' and '1.'

STORING COMPUTER DATA/PROGRAMS ON AUDIO MAGNETIC TAPE

To reverse the process, one simply plays back the recorded tones through a device that converts the two different audio tones into appropriate logic levels. The two logic levels derived are used to represent the two possible states of a binary bit. This information is fed back into the computer in a serial fashion to build up eight bit words which may be stored back in the computer's memory banks.

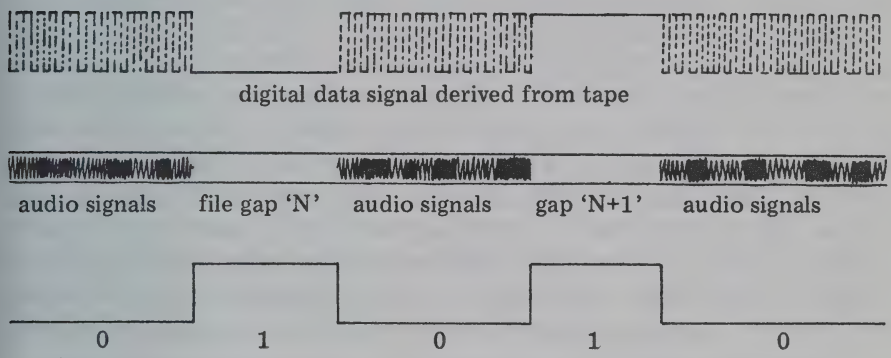
With an appropriately designed tape recorder interface circuit, and a tape recorder having the proper control options, it is possible to place the entire recording and playback process under control of the computer. When a computer program is instructed to transfer information to magnetic tape, the computer can send a signal to start the motor on a tape recorder (to get the magnetic tape moving properly), and then after waiting a few seconds or so (to make sure the tape is in motion), send the data that is to be stored on the tape. As was the case with using paper tape, binary patterns representing instructions, data, or addressing information (telling where in memory the information is to be placed on playback) may readily be stored on the magnetic tape. When the necessary data has been transmitted from the computer to the tape unit, the computer can send another signal

to stop the tape motion motor. In sophisticated systems, additional control signals may be used to, for instance, rewind the tape on a cassette or cartridge, or place the tape unit in a "fast forward" mode so as to quickly skip over sections of a tape.

The kinds of extra capabilities being discussed for a magnetic tape unit are important because the compactness of magnetic tape makes possible procedures that are not practical when using punched paper tape. Earlier it was mentioned that is is possible to save several hundred thousand bytes of information on a single magnetic tape cassette. For illustrative purposes, assume that figure was based on storing 100,000 bytes on each side of a cassette. That means, once installed in the tape recording/playback unit, the cassette could store 100,000 bytes of information along its length without human intervention.

Now 100,000 bytes can represent a lot of programming directives, or a lot of information serving as data for a program. That is far more data than a typical microprocessor can hold or digest at any one time, and far more instructions than could be held in a typical microcomputer's memory banks! In fact, for practical purposes, if the tape was to hold data for processing by a program, chances are that for practical programming purposes it would be organized as chunks of data consisting of from a few hundred to a few thousand bytes at a time. Or, if sequences of instructions representing programs were being stored, it is more likely that they would be of similar length. What does this mean? Simply that a cassette can usually hold 10, 15, 20 or more blocks of information along its length. Blocks of information on a mass storage medium are often referred to as FILES. This is because they may be conceptually viewed as being similar to packets of information that one might segregate into folders in a paper filing cabinet. The files of information stored on a magnetic tape must be separated from one another in some manner, just as manila or cardboard separators are used to separate files in a paper file drawer. However, since magnetic recording does not leave any visable marks on the tape, some other way of marking the end of one file and the start of another has to be established. The method generally used is quite simple - one simply does not record anything between blocks of information, thereby creating what is commonly called in computer circles a FILE GAP. Thus, in the simple audio tone recording method described, one would stop sending any audio tone entirely

while the tape continued running for a few seconds to create a "gap" on the tape where no sound would be recorded. Now, using appropriate electronic circuits, it is possible to design the tape unit interface so that, for instance, when an audio signal is present, a special "file gap" control signal will be in a particular logic state (such as zero). When no audio signal is present, the control signal will switch to the opposite state (such as logic one). This control signal can then be connected to the input of a computer so that a computer program can count the number of file gaps detected. By counting file gaps, the computer can be programmed to automatically locate particular files on a long tape cassette!



PICTORIAL OF FILE GAPS ON MAGNETIC TAPE

It must be pointed out that recording through the use of audio tones is by no means the only method by which data may be recorded on magnetic tape. In fact, the use of audio tones on low cost audio tape recorders for the storage of computer data did not really become fashionable until the advent of personal computer use and the subsequent demand for low cost magnetic tape storage capabilities. There are much more sophisticated magnetic tape storage techniques that have been in widespread commercial use for many years. These

techniques are considerably more reliable than the audio tone method. They are also generally more expensive - both for the recording/playback equipment, and for the high grade "digitally certified" (tested for continuous quality the entire length of the tape) cartridges or cassettes of magnetic tape recommended for use with such equipment. These magnetic tape units designed specifically for use with computers operate through use of a technology known as saturation recording. This method relies on the technique of magnetizing tiny segments of the recording medium in two opposing directions (one direction at a time!) with each magnetized segment representing a binary bit of information. When the magnetic tape is played back, pulses are generated by the magnetized segments which are decoded to represent logic states. The number of binary bits one can reliably place on magnetic tape using such techniques is often referred to by the abbreviation BPI, which, logically enough, stands for Bits-Per-Inch! Naturally, the higher the BPI figure for a tape unit the more data one can pack on a given amount of tape. Furthermore, a higher BPI figure generally correlates with a higher transfer rate. In days past, digital tape units often boasted a 200 or 400 BPI capability. Today, figures of 800 to 1600 are common, and much higher figures can be achieved in premium units. It is interesting to note, though, that as in low cost audio recording units, file gaps (short lengths of tape with no magnetic pulses) are still used to separate files on the magnetic tapes that use saturation recording methods.

While talking about terminology associated with input/output operations and mass storage devices, it might be interesting to learn about a measure of reliability/quality that one is likely to see or hear mentioned frequently. The measurement being referred to is known as the bit-error-rate, and is simply a figure indicating how many bits are likely to be in error when data is transmitted from one device to another, or, for instance, played back after being recorded on a medium. The figure is normally expressed as the number of errors likely to occur divided by the total number of bits in a representative sample. For instance, if one read that the figure for an error rate is 1×10^{-5}, it would mean that the device being referred to had an error rate of 1 wrong bit out of every 100,000 bits processed (1/100,000).

In a magnetic tape system, errors can be caused by faulty mechanical equipment, "noise" or "disturbances" in sensitive electronic cir-

cuits, tiny particles of dust on the magnetic tape, imperfections in the magnetic coating on the medium, and similar types of defects. The higher the quality of the system, the more reliable it should be, and the lower one may expect the error rate to appear. Magnetic tape systems designed around audio tape recorders may be found to have error rates in the vicinity of 1×10^{-6} (not so good) upwards to 1×10^{-8} (rather decent for the technology). Commercial grade digital magnetic tape units can reach error rates as low as 1×10^{-10}.

Quite often when referring to bulk storage devices, error ratings will be given as two figures. One is called the SOFT ERROR RATE. This error rate refers to errors that may occur due to temporary anomolies, such as a piece of dust causing a bit to be misread, which, if the read operation is repeated, may result in correct information being recovered. The other measurement is referred to as the HARD ERROR RATE, and refers to errors that will show up permanently. Such errors may be related to defective medium (magnetic coating), invalid or unreliable recording, and similar problems. When both measurement figures are provided, the soft error rate invariably indicates poorer performance than the hard error rate. In practice, the soft error rate gives an indication of how often one might expect to have to reprocess information in order to obtain valid data. The hard error rate - indicating non-recoverable loss of data - is the really critical figure in determining overall unit performance.

While bit error rate figures are important for determining a merit of performance for various pieces of equipment, it should be realized that such figures do not necessarily dictate absolute system performance. This is because various error-correcting techniques may be employed in communicating (data-transmitting) systems to recover information lost at the bit level. Such methods, if they do not recover the information absolutely, provide indications that will lead to recovery from a practical applications viewpoint. (For instance, if the ASCII code is being used to transmit characters in words composed of English text, and a bit state is reversed causing an incorrect letter to be displayed. The position of the letter within the word coupled with the context of the word within a message can generally lead to correction of the error by a person reading the message.)

This section has pointed out that dollar for dollar, a magnetic tape

device provides higher performance as an external mass memory device for a computer system owner over a paper tape system - unless there are special circumstances that might render the magnetic system ineffective, such as hostile environment. Does this mean that magnetic tape units are the ultimate devices to use for bulk storage? Not necessarily! Magnetic tape as a mass memory medium has a major drawback in that the information on the tape must be serially accessed. That means one must traverse intervening sections of the tape of get to a particular point of interest. If, for instance, there are 20 blocks of data (files) on a tape cassette, and one needs to obtain information stored in file number 20 when the tape is positioned at file number 1, it is necessary to wind through the entire tape in order to get to the appropriate file. This process, even if a high speed "fast forward" mode is provided on the tape unit, takes considerable time. Typically, the time taken to locate a desired file of data on a tape cassette or cartridge can be from tens of seconds to several minutes. That is a relatively long time when one considers that once the information is located on the tape it may take only a few thousandths of a second for the information to be transferred to the memory banks of a computer and processed.

The amount of time it takes on the average for a mass memory device to access a given record of information (file) chosen at random, when the device starts from a random starting point, is designated as the AVERAGE RANDOM ACCESS TIME. The average random access time for a serially formatted bulk storage device may be derived by calculating the length of time it takes for a WORST CASE ACCESS and dividing by two. The worst case access time for a magnetic tape unit would be the time it takes to traverse the entire length of tape stored in a cartridge or cassette.

As an example, consider a digital magnetic tape system that uses cartridges containing 150 feet of tape. Assume that in the normal record and playback mode the tape moves at five inches per second. But, in the fast forward or reverse mode (when the equipment does not transfer data but simply looks for and counts file gaps) that the tape moves at the rate of 60 inches (5 feet) per second. Using these figures, the worst case access time (to go from one end of the tape to the other) would be 150 ÷ 5 = 30 seconds. One half this value would be 15 seconds - which would be the average random access time of

the unit. That is, on the average, the time to get from one file number (chosen arbitrarily at random) to a second file (chosen in the same manner) would be 15 seconds. Sometimes the actual time required would be 30 seconds, other times less than a second - but on the average, 15 seconds. (This example is typical of what a fairly good digital tape unit would be capable of performing.)

The above time might not seem so slow to a newcomer to computer technology, but look at how that kind of time might add up in a typical computerized application.

Suppose the example tape cartridge containing 150 feet was formatted so that it contained 450 files. For hypothetical purposes, suppose each file required 3 inches of tape and contained a name and address of a company, along with some billing information. (To make things come out nice and even, assume a file gap between files of one inch!) Now suppose it was time to produce a monthly statement, and for whatever reasons, all of the accounts had to be randomly accessed twice during the operation. Fifteen seconds, times two, times 450 comes out to 13,500 seconds. That would be three hours and forty-five minutes spent just accessing information by going back and forth across that tape!

In summary - magnetic tape systems do not have impressive random access times. Are there ways to circumvent this shortcoming? Sure. People have been working on reducing the magnitude of the problem for years. One way to reduce the effects of serial access is to carefully organize the information stored on tapes so that a minimum of random accessing needs to be done. Frequently used files can be placed close to one another. Another way is to shorten the lengths of tapes and increase the number of tape units. Thus, if the 150 foot tape previously mentioned for discussion purposes was cut in thirds so that there were three 50 foot cartridges, installed in three tape units (all under control of the computer), the average random access time would be effectively reduced to 5 seconds - down from 15 seconds. The type of operation that formerly required three hours and forty-five minutes would be reduced to one hour and fifteen minutes. That is progress.

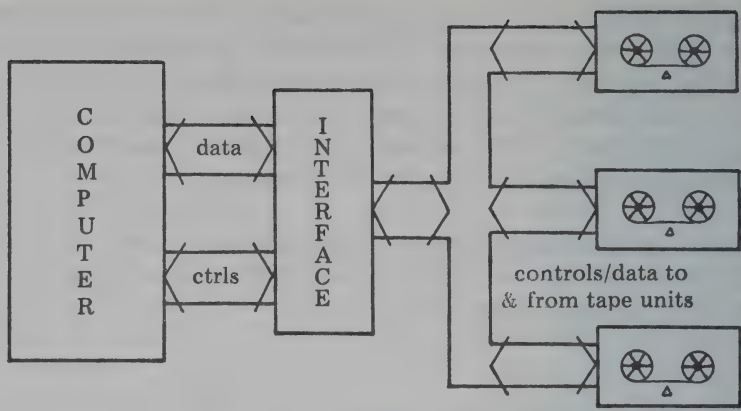
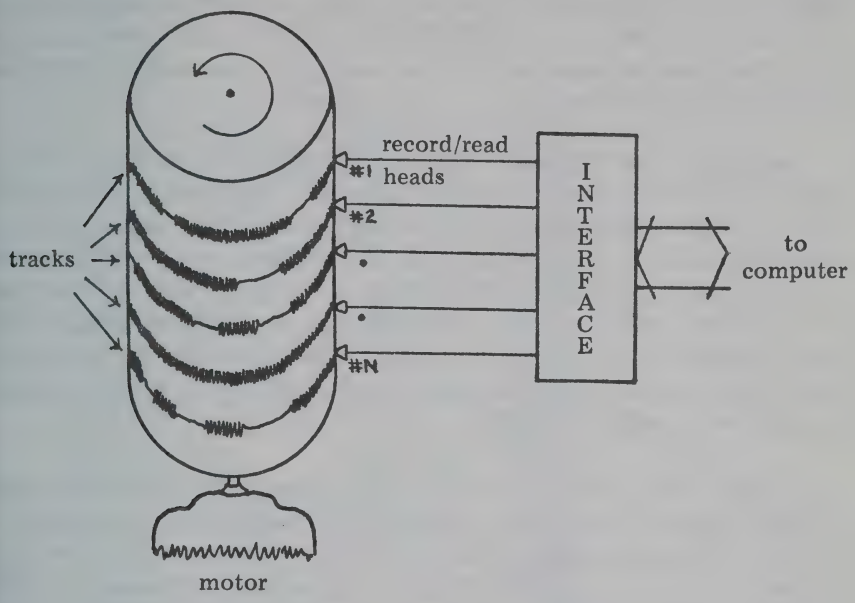


DIAGRAM OF MULTIPLE TAPE UNIT SYSTEM

Suppose the original 150 foot length of tape was cut into 150 one foot sections, and placed on 150 cartridges in 150 tape units? The average random access time would now be down to just 1/10 of a second! But, the system would be rather expensive with all those tape transport units! Ahah - but the concept is intrigueing. Why not, instead of putting the one foot lengths of tape on individual cartridges, wrap all the one foot lengths around a single cylinder so that only one mechanical tape unit was required. Each loop of tape formed by wrapping it around the drum could be assigned its own tape read/write head. Spinning the drum on its axis would result in all the circles of tape simultaneously being scanned by their own recording/playback heads. The average random access time between any two files on the loops of tape anywhere on the drum would be equal to the amount of time it took the cylinder to rotate divided by two. (Which is still the worst case access time (one rotation of the drum) divided by two as was the case for lengths of tape on a cartridge.)It is interesting to note, that even if the rotational direction of the drum is never changed, the average random access time between files is determined by the same formula as was used when a length of tape was stored on a cartridge that was able to reverse the direction of the tape. The reason is of course, that while the direction of the cartridge tape motion could be reversed, when the end point of a tape is reached, the tape must be rewound to get to the start of the tape.

Whereas, the end point on a circle of tape wrapped around a drum joins the starting point. Thus, going from the end back to the beginning does not require a reversal of drum motion.



A cylinder coated with magnetic material (similar to that used on magnetic tapes) is spun at high speed on its axis. Record/playback heads are arranged along the length of the drum. As the drum spins "tracks" of magnetic material are spun past the heads so that data may be magnetically recorded or recovered.

MAGNETIC DRUM MASS MEMORY DEVICE

The type of device just described has indeed been in use for many years as a mass storage device for computers. The devices are called DRUM MEMORIES, and in the early years of digital computers, these devices were the ultimate in mass memory performance. Large drums, some several feet in diameter, were coated with magnetic material. Magnetic recording/playback heads were placed along the length of the cylinder so as to make as many as a hundred or more tracks around the drum on which data could be recorded. Digital

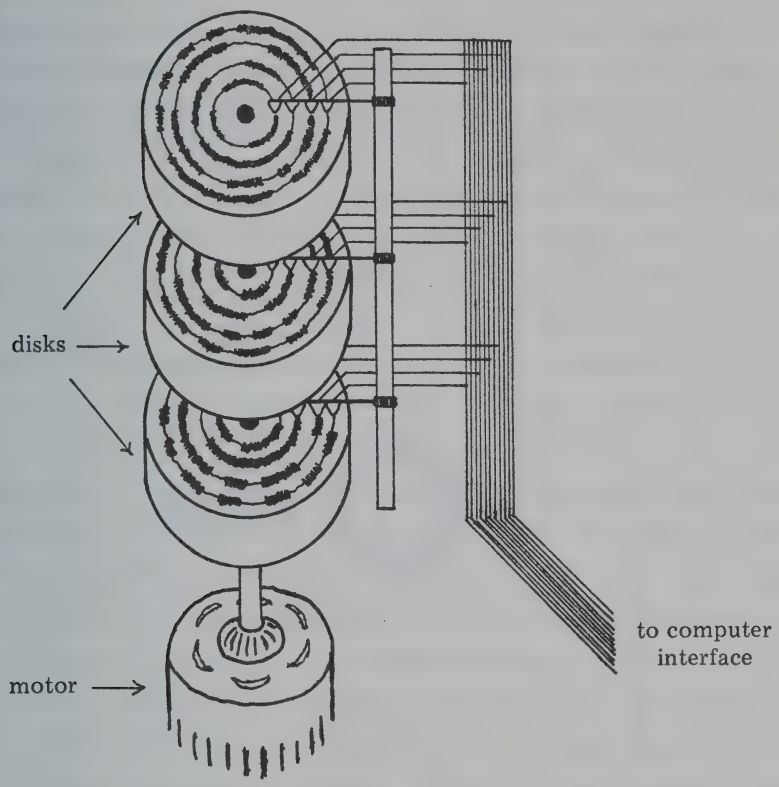
recording techniques were generally used (where tiny segments of the magnetic medium were polarized in alternating directions to represent binary states). The drums were spun on their axes at quite high speeds. These devices were expensive (often tens of thousands to hundreds of thousands of dollars in cost). However, larger models could hold upwards of 25,000,000 bits (3,000,000 bytes), and the drums had fast average random access times measured in the thousandths of a second range!

It soon became apparent that the concept of spinning a path of magnetic material past a magnetic recording/playback head need not only be done along the cylindrical surface. It could also be accomplished by using the ends of the cylinder. In fact, one could do better by cutting up the cylinder into platters, coating the tops and bottoms of the platters with magnetic material, placing banks of data-passing heads across the surfaces of the platters so as to form circular tracks along those surfaces, and spinning the platters! The platters looked like, and were called, DISKS!

The disk approach allowed still more data to be stored in a given amount of space. Some large disk units of today can store as many as 160,000,000 bits of information (20,000,000 bytes). (That is enough storage space to hold about 250,000 names and addresses in ASCII coded form!) These units, somewhat beyond the pocketbook range of individual consumers, are used in business applications. They, as did drum units, have average random access times measured in the thousandths of a second range!

As some readers may well imagine, the magnetic read/write heads used in digital tape systems and drum and disk units are not exactly cheap. The consequence of putting a few hundred of these heads into a disk or drum unit in order to simultaneously be able to scan all the tracks provided on such units is a high price on the unit. Why not, on a disk unit for instance, mount a single read/write head on an arm that could traverse the surface of the disk (much as the arm on a record player can reach any groove) so as to reach any desired track? One such head-equipped arm unit per surface would eliminate a large number of expensive recording/playback heads. True, there would be some lowering of performance for the unit. If the information being sought was not on the particular track that the arm holding the head

was currently positioned to, the arm would have to be positioned to the track where the desired information was stored. This would take some extra time, but relatively little time compared to, say, the average random access time of a magnetic tape unit. The compromise in total unit cost versus performance seemed worthwhile. Disk units,



If a magnetic drum is cut into platters, the top (and often the bottom) surface of each platter may be used for recording purposes. Since a cylinder may be sliced into many platters, more recording surfaces are provided per unit of volume than when just the surfaces along the axis of a cylinder are utilized. Record/playback heads are arranged along the platter surfaces to record/recover information from the circular tracks as illustrated.

MAGNETIC DISK UNIT

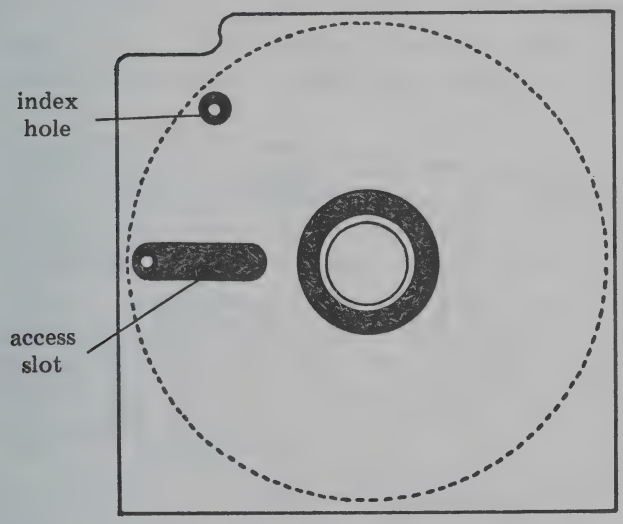
equipped with an arm (or arms in multiple spindle units) that can move a recording head to the desired track on a disk surface are called MOVING HEAD DISK systems. The prices of these types of units came into the thousands of dollars range. Think that was high? The FIXED HEAD DISK systems using multiple heads were generally priced in the tens of thousands of dollars range.

Thousands of dollars for a mass storage unit, even if it is relatively fast, still puts the unit out of any so called "mass-market" range. The problem with regular disk units is that they are devices requiring expensive, close-tolerance machining and manufacture. Why? Because the disk surfaces spin very fast - typically thousands of revolutions per second. These surfaces are coated with a very thin layer of magnetic material that is easily scratched and/or worn away. In order to prevent the magnetic layer from being quickly destroyed as the tracks are scanned by the head(s), these units are precision manufactured so that the disk surface does not actually touch the recording head. They come within a quarter of a millionth of an inch or so (typically) of contacting one another. They are actually designed so that the effect of the air swirling along the surface of the disk keeps the head from touching the surface. Naturally, this type of operation requires precise manufacturing tolerances and high quality workmanship, which increases the price of the product. The nature of these devices also requires that they be maintained in clean, vibration free environments. Tiny dust particles finding their way into a high speed disk unit can quickly destroy a disk surface under the action caused when they attempt to pass through the microscopic gap between a head and a spinning disk. Jarring an operational disk unit can cause a head floating on the cushion of air created by the spinning disk to bump into the platter surface. Such a collision between the head and high speed rotating disk can cause a section of magnetic material to be scraped from a track causing irrepairable damage to the unit. (Such collisions are commonly called DISK CRASHES in the computer field. If it causes the loss of vital system operating data from a section on a disk, a big expensive computer system can be reduced to a useless group of blinking boxes. The term can cause terror in the hearts of data system managers.)

Obviously, it might not appear that disk units would be ideally suited to small computer systems in environments such as homes,

shops, and small businesses, where strict disk-protective measures might not constantly be observed. Fortunately for small system enthusiasts, advances in technology continue to bring forth new findings, new ideas, and new solutions to problems.

Not too many years ago, someone got the idea of creating a flexible disk with a relatively tough coating of magnetic material. They coupled it with the idea of spinning the disk at a much slower speed so that a recording head could contact the medium without causing excessive wear. The resulting product has become known as a FLOPPY DISK UNIT. The name is derived from the flexible medium on which the data is recorded. It is supplied in the form of a jacketed disk

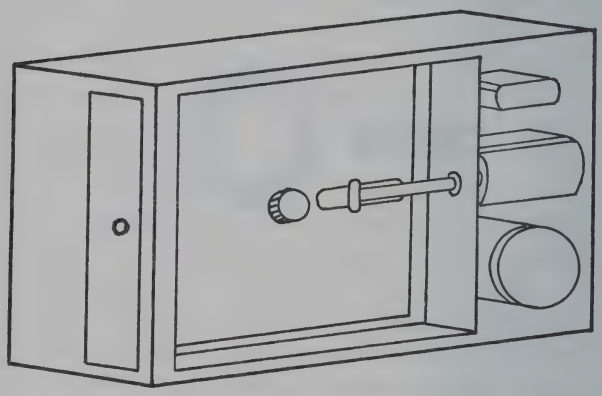


The record-like platter on which information is recorded in a floppy disk unit is enclosed in a protective sheath. An access slot in the sheath allows the record/read head to contact the medium when data is being recorded or recovered. An index hole along the perimeter of the platter allows photo-electric techniques to be used to determine a starting reference point for all the tracks on the unit. The platter in its sheath is often referred to as a "diskette."

FLOPPY DISK RECORDING MEDIUM - "DISKETTE"

similar in size to an ordinary 45 RPM platter commonly used fo merchandising popular music.

Floppy disk units turn out to be nice compromises between stan dard mass memory disk units costing thousands of dollars and lo cost digital tape units costing several hundred dollars - both price-wis and performance-wise. Today, standard floppy disk units retail in th 1,000 dollar price range. They are capable of storing approximatel 250,000 to 500,000 (double-density versions) bytes of information Typical average-random access times from any location on the dis to any other location is generally less than a second. (Track-to-trac accessing requires moving the read/write head-carrying arm.) Th disk typically spins at three to four hundred revolutions per minut giving a single-track average random access time of in the vicinity o 1/10 of a second.



A floppy disk (diskette) is inserted through a slot or small door opening into the floppy disk unit. Once in the unit, the platter is spun (inside its protective sheath) by a motorized spindle mechanism which engages and aligns the platter through the large hole in its center. Electronic and mechanical controls position an arm holding a record/playback head against the diskette (along the access slot) at the proper track to read or record data.

FLOPPY DISK UNIT

Once a particular file has been located on a floppy disk, information can typically be transferred to or from the computer memory banks at rates running as high as 25,000 to 50,000 and more complete bytes per second. With the relatively low random access time compared to tape units and high transfer speeds, the floppy disk unit is fast becoming a highly popular mass memory device for the serious microcomputer enthusiast. The recent advent of so called mini-floppy disk units, which have about half the storage capacity of the standard units through use of a smaller disk, and a price tag about half that of standard sized units, should further increase the popularity of these devices.

Besides the specific types of devices discussed that may be connected to a computer to allow an operator to input information in the form of characters, and receive like data, or that are used as mass storage devices, there are countless types of unique devices that one may couple to a computer through appropriate interfaces.

Some kinds of devices may be extremely simple. These could include manual electrical switches that, depending on their toggle position, would either present a digital logic one or logic zero condition to the input lines of a computer. Or, the switches might be automated as part of a piece of machinery. For instance, a momentary action switch operated by a cam on the shaft of a motor might be used to signal a computer each time the shaft was rotated. This arrangement could provide the means for a computer to count the number of revolutions, or frequency of rotation.

Going in the other direction, from computer to external devices, the voltages associated with logic levels might be used directly to activate simple devices - such as Light Emitting Diodes (LEDs) or seven-segment and similar alphanumeric displays. Or, the signal from the computer might drive electronically activated switches and devices such as solenoids. Alternately, the low voltage levels from the computer might be fed to relay coils that would in turn operate switches capable of handling high voltages while keeping them isolated from the computer. In this manner, the computer may be arranged to control devices that, for instance, run on ordinary house current, such as electric motors, toasters, robotic devices, etc.!

Quite often, however, it is not sufficient to simply use binary logic levels to control external devices. (Binary logic levels being signals that are either in a logic one or logic zero state, or pulsing between the two conditions.) Many devices and instruments require (or produce) electronic signals that are constantly varying over a wide range of levels (instead of switching between essentially two constant amplitudes). Such signals are called analog signals. There is a basic incompatability between digital computers and analog signals. Digital computers are designed to operate with and only properly recognize two levels of signals representing the two possible binary values (0 and 1). They cannot properly react to analog signals that may vary between the recognized levels to which the computer responds.

Fortunately, over the years electronic engineers have developed devices called Analog-to-Digital (abbreviated A/D) converters, and their counterparts, called Digital-to-Analog (D/A) converters. These devices work by changing signal levels to binary patterns, or vice versa. The amourt of resolution that can be obtained using an A/D converter is partly a function of how many binary bits are used to represent an analog value. Eight bit converters are frequently used with microcomputers, but 10, 12 and even 16 bit resolution devices do exist (at increasing expense).

Vector generated graphic displays that produce line segments connecting points on a cathode-ray-tube display use D/A converters to convert binary patterns sent by the processor to analog voltages that control the electron beam on the face of the tube.

Analog-to-digital converters are frequently used to convert information on a X-Y digitizing table to a form that can be used by a computer. Conversely, Digital-to-Analog units may be used to drive X-Y plotting devices that can draw maps, diagrams, pictures, and so forth under the direction of a computer. Digitizing tables and X-Y plotters are frequently used to enable a computer to assist in the preparation of complex manufacturing drawings. For instance, using a digitizing table as input, a person can prepare a rough layout of component placement for a printed circuit board using actual components so that sizes can be accurately specified. Connection points are identified as parts are positioned, and those junctions correllated with other information, such as points on an electronic schematic diagram

(that shows how components relate to one another electronically by electrical pathways). A computer can then be programmed to produce a printed circuit wiring diagram and display a suggested routing pattern for all the electronic paths on the circuit board by drawing the computer generated pathways using an X-Y plotting machine.

The types of actual I/O devices that may be connected to a computer are virtually endless, and limited only by the imagination of the system designer coupled with practical considerations. As suggested in this chapter, one may divide I/O devices into three major classes. Those that allow an operator to conveniently communicate with a program in the computer. Those that serve as bulk memory storage devices. (These allow large amounts of data and programs to be transferred from the "active" form in the read and write memory of the computer to a "dormant" form on an external memory medium. When needed to be activated or manipulated further by the computer, the information they hold can be quickly loaded back into the main memory of the system.) Finally, there are those I/O devices that do not fit into the first two categories. These range from the simple to the complex (such as devices that may be used to accept measurements from instruments or machines, or send control signals to such devices).

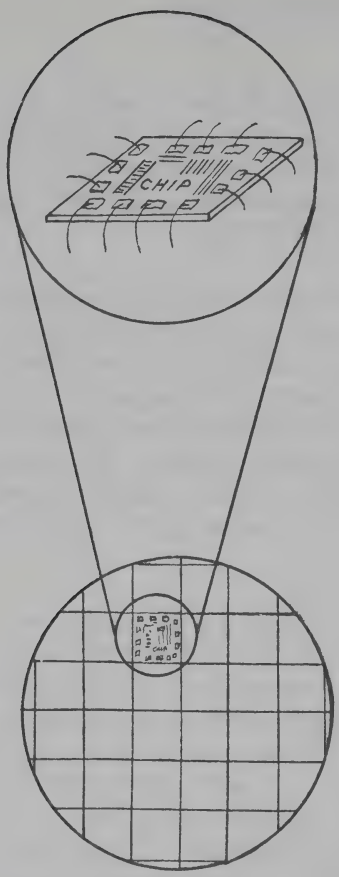
In essence, I/O capability turns the computer into a practical, useful machine. By itself, the computer only has the capability to manipulate vast amounts of information and perform complex calculations at high rates of speed. With I/O devices, the marvelous capability of the machine may be directed and controlled conveniently by those it is designed to serve. Its appetite for assimilating, correllating, and condensing data may be properly nourished. The results of its work can be unleashed to control machinery and enlighten its masters.

SYSTEM CONSIDERATIONS

The reader has now been introduced to the concepts of a microcomputer's operation, how it may be programmed to perform tasks, and how it needs I/O accessories in order to perform useful functions. It is time to consider some practical aspects associated with constructing or assembling a microcomputer SYSTEM.

Before getting into specific types of systems that one might want to consider putting together in order to perform various types of tasks, it might be beneficial for some readers if a few matters relating to the physical construction of an electronic computer system were discussed. Such a presentation might provide some insight into the practical aspects of setting up a computer system. It will also shed some light on how one may expand an initially small computer system with very limited capabilities into a larger system with enhanced processing power.

Electronic computer systems of today are primarily based on a technology known as LARGE SCALE INTEGRATION (LSI) of electronic circuits. What this means in essence, is that large numbers of individual electronic components, each having very limited capability (such as being able to "remember" a binary value), or when combined with each other, are able to perform a very simple logic function (such as the AND or OR operation mentioned in Chapter Two), are manufactured using microminiature techniques. This process enables thousands of these components (known as transistors and diodes) to be crammed into spaces measured in terms of square millimeters. These circuits are manufactured on material (typically silicon, a nonmetallic element) which has been "doped" (blended with an impurity) with a small amount of metallic substance, resulting in a compound that is called a SEMICONDUCTOR. The material containing the microminiature components typically ends up in the shape of a thin square or rectangle. These squares or rectangles, measuring from fifty to several hundred millimeters on a side (0.5 to 2 centimeters) are often referred to as CHIPS. The expression "CPU-on-a-chip" comes from the fact that typical microprocessor CPUs capable of performing an entire repertoire of instructions (such as that presented in Chapter Two) are able to fit on a single "chip" of semiconductor material.

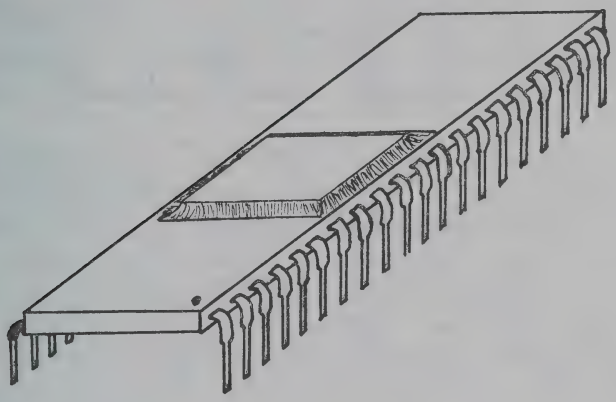


Integrated circuits are manufactured using microphotography techniques. Ultra-miniature components are chemically built-up on thin slices (wafers) of silicon material. Each wafer provides space for many identical "chips." After processing the wafers are divided into individual "chips." Tiny wires are then attached to microscopic bonding pads on the chip. The chip will be placed in an IC package. The other ends of the wires will be connected to metal pins extending from the sides of the package. The chip will then be hermetically sealed inside the package for long term protection of the delicate electronic circuitry. The completed device is referred to as an integrated circuit or IC.

SILICON WAFER AND INTEGRATED CIRCUIT CHIPS

Individual chips, such as a microprocessor CPU, are packaged inside small INTEGRATED CIRCUIT packages. Microscopic wires are connected from various connection points on the integrated circuit chip to larger metal pins that generally extend from the sides of the packages. The number of wires necessary to bring out the various electronic signals that may be present on the semiconductor material (or, for that matter, carry signals into the chip) varies with the complexity of the device contained in the package. Many CPU devices

have about forty pins for carrying the various electronic signals between the "outside world" and the semiconductor chip installed inside the package. These pins, extending from the sides of the INTEGRATED CIRCUIT (IC as it is frequently abbreviated) package, allow the device to be plugged into a socket or soldered to wires connecting the IC with other parts of a computer.

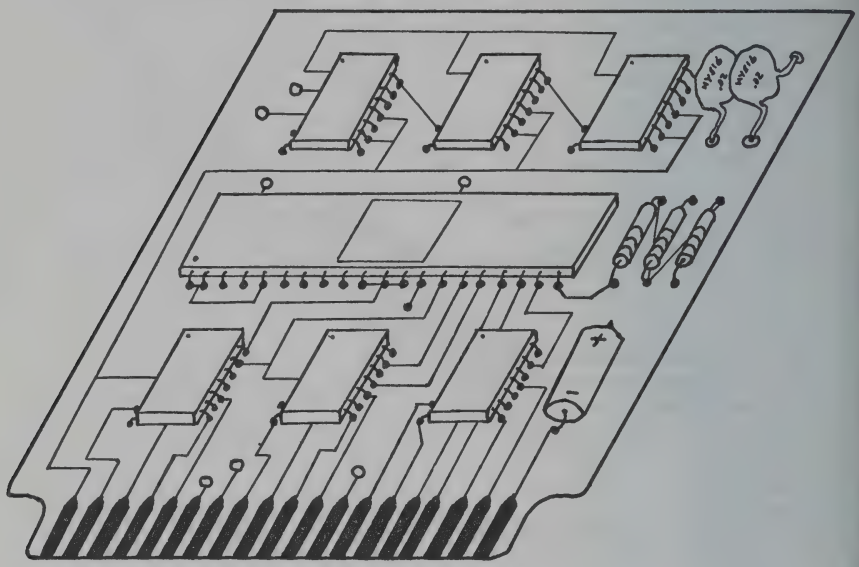


A 40 PIN INTEGRATED CIRCUIT PACKAGE

In addition to a CPU-on-a-chip, microcomputer systems have many other types of integrated circuits. Memory elements, special logic and control circuits, interfaces to peripheral (external or auxiliary) devices, are also frequently manufactured in the form of integrated circuits. A typical microprocessor system may contain from just a few to many hundreds of individual integrated circuit packages.

These devices must all be properly hooked together in order to form an operational computer system! This means that pins extending from one integrated circuit package must be connected to pins on other IC packages. This can be accomplished by running wires between integrated circuits. Frequently, however, instead of using individual pieces or wire, connections between pins on integrated circuits are made via metallic pathways formed by special manufacturing

processes. These electrical pathways generally consist of copper metal that has been deposited (plated) upon a rigid board-like material made of phenolic or glass-epoxy material. The resulting creation is commonly referred to as a printed circuit board or printed circuit card.



Integrated circuit packages and other electronic parts are typically mounted on printed circuit cards. Metal pathways on the card connect various components to one another and interconnect pins on integrated circuit packages. Cards (such as the one illustrated) are often designed to be plugged into "printed circuit card sockets" along one edge. This allows printed circuit cards to serve as modules in a modular system.

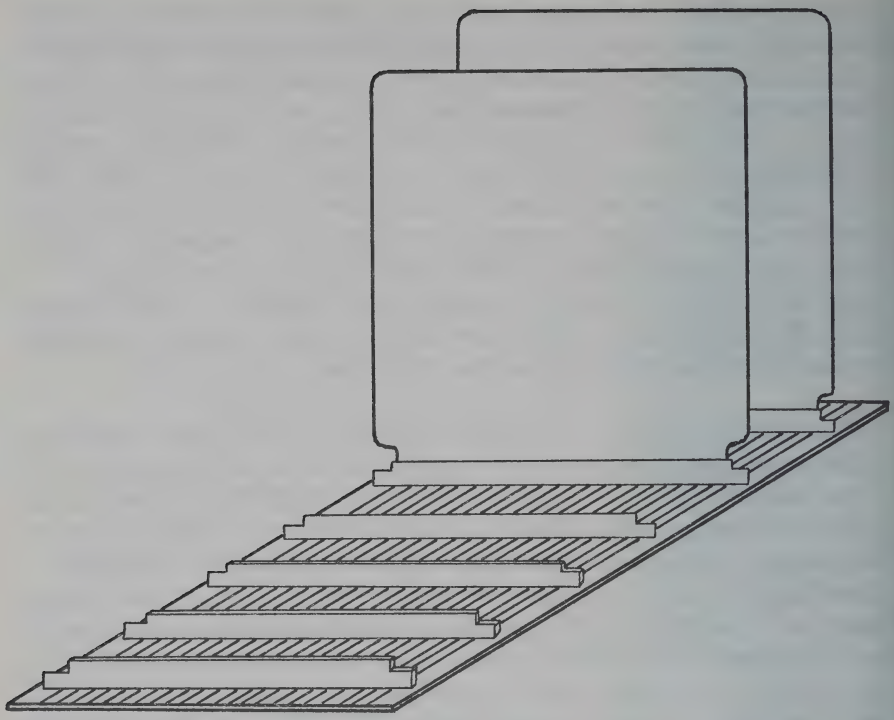
PRINTED CIRCUIT CARD

Quite often, the particular types of integrated circuits mounted on individual printed circuit cards will be segregated into functional categories. For instance, one type of printed circuit board may contain components and integrated circuits relating more or less exclusively

to the central processor unit and associated control circuits. (Generally, a so-called "CPU-on-a-chip" requires some supporting circuits to make it operational in a practical computer system. Names the reader is likely to encounter for circuits in this category include CLOCK DRIVERS, BUSS BUFFERS of various categories such as ADDRESS, DATA and CONTROL BUFFERS, and BUSS CONTROL ICs.) Another type of printed circuit board may contain electronic circuits that are used to form memory elements (organized as 8-bit words) and corresponding "addressing" logic that enables the computer to specify individual groups of cells in groups of 8 bits (bytes). Still another printed circuit card may contain circuitry primarily related to controlling input/output operations.

The technique of grouping various types of functions together on individual cards allows a microprocessor system to be constructed in a modular fashion. A large system may be built up from small modules having various kinds of functional capabilities. There are economies in design and production using this technique resulting in advantages to the end user or consumer. For instance, if the memory banks of a computer can be built up from small modules, then a user can purchase just the amount needed for a particular application. Or, as is often the case in actual practice, one may conveniently add extra sections of memory elements as one finds the need for expanded capability.

Modules - the circuitry contained on a single printed circuit card - must be interconnected among themselves in order to form a complete computer. Quite often, printed circuit cards are designed so that they may be plugged into special types of sockets called printed circuit card edge connectors. A typical interconnection arrangement is often provided for constructing a large printed circuit board called a BACKPLANE that contains a number of BUSSES running to a number of printed circuit board edge connectors. BUSSES are simply electronic pathways (wires or the equivalent) that join a number of connectors (modules) together. Quite often the busses on a backplane are grouped together and referred to by their functions or the types of signals that they carry. Most microprocessors have, for instance, ADDRESS BUSSES, DATA BUSSES, CONTROL BUSSES, I/O BUSSES, and so forth.



A backplane board containing busses is commonly used to build up a computer system. Printed circuit card sockets are installed on the board thus allowing printed circuit card modules to be connected to the commonly used system signal lines. Such a modularized system is readily expanded or upgraded by simply plugging in additional modules as the need or their availability arises.

BACKPLANE BUSS BOARD

Buss structure is an important parameter to consider in a microprocessor system. There are different ways to design microcomputers and develop buss architectures. A key advantage of a buss arrangement is that it allows system flexibility. A well designed buss arrangement will allow any type of module to be plugged into any circuit card connector on a backplane where it can pick up the signals it needs (and deliver those it produces) from (to) the appropriate electronic pathways. A "standardized" buss arrangement makes it possible

for different manufacturers to produce modules that can be interchanged. Or, for manufacturers with different kinds of technical expertise to provide modules with unique capabilities that may be used in any system having a particular "standardized" buss structure. There are attempts being made to standardize on buss structures (by defining the layout and purposes for each signal path in a particular physical backplane arrangement). However, there is no single plan which has gained so-called "universal acceptance." A detracting factor in this effort to standardize from a manufacturer's point of view is that such standards can make it easier for competitors to sell competing modules. For that, and other reasons, a number of manufacturers prefer not to produce computer systems that utilize standardized buss structures.

In any event, one thing a prospective small computer system owner should consider is whether or not the system to be acquired utilizes a standardized buss arrangement. (If it is big enough to use any backplane network at all. Some small systems, it should be pointed out, are not designed to be expanded and, hence, do not utilize any such buss arrangement.) If the system does utilize a buss, the user might want to inquire as to whether the buss arrangement can serve various types of CPUs, memory modules, I/O interfaces, etc. It turns out, for instance, that some buss conventions allow CPU modules to be replaced with modules containing altogether different types of CPU devices, as well as CPU supporting modules (memory, I/O, etc.), while other buss structures may only permit one type of CPU device to be installed while retaining flexibility for interchanging supporting modules.

The reason for considering whether a buss structure will support different types of CPU devices is related to the CPU's primary importance in a system. The signals that it requires and that emanate from it determine the relationship between it and all the other supporting modules in a typical computer system. The CPU is the workhorse or powerhouse of a computer. It determines the fundamental capability of a particular computing system. A great deal of work is constantly being done towards improving the performance of CPU devices. Manufacturers are constantly finding ways to upgrade their devices and announcing new units with new capabilities in terms of instruction sets, speeds of operation, lower cost, etc. Different CPU devices

produced by various manufacturers have varying capabilities. Some units are designed with instruction sets that make them more useful in, for instance, text handling (working with alphanumeric messages) capability. Others are designed to serve the needs of those desiring to perform mathematical calculations to a great extent. Still others may be tailored for use in machine control applications. (Note: the typical microprocessor as described in this book is capable of performing all the kinds of manipulations being mentioned here. What is being pointed out is that some CPU devices are specifically designed to be a little more powerful in one area than another might, depending on the designers goals.) In short, CPU devices can become outdated, or a particular CPU may be considered inefficient in a particular application. Accordingly, it sometimes becomes desirable to replace the CPU module in a system with one containing a different type of CPU! However, regardless of what type of CPU is being used, the rest of the modules in a typical microcomputer system can still be used! Virtually any microcomputer needs memory elements, I/O interfaces and so forth. Thus, a buss structure that allows CPU modules containing different kinds of CPU devices to be plugged into a system has a considerable advantage over a system that does not permit such a change!

Note should be made of the fact that while many manufacturers produce systems that capitalize on a backplane with a buss structure that allows modules to be plugged into it - they do not necessarily use a buss structure that lends itself to easy interchanging of CPU types.

Since the CPU being utilized in a system is the dominant device many manufacturers name or label their computer systems after the type of CPU device it is based upon. CPU devices are commonly referenced by numeric, alphabetical, or alphanumeric nomenclature. The nomenclatures assigned to CPU devices are really just style or model numbers more or less arbitrarily assigned by the developers. These names typically stick, and are promoted by the manufacturers to identify their devices. Since there are numerous manufacturers many of which produce several different types of CPU integrated circuits, the number of such CPU designations is already large and proliferating rapidly. When reading the trade literature, one will see references to CPU device types such as 8080, 6800, 6500 series, and

so forth. The reader must realize that each such type of device has different characteristics. Different types of CPUs have different kinds of instruction sets, manufacturers generally assign each instruction set its own mnemonics, and each CPU generally recognizes a unique set of machine language codes. The actual details of each device's capabilities must be studied from the manufacturer's literature and specification sheets.

The proliferation of different types of CPU devices poses problems for users who want to develop application programs that are most efficient if the machine is programmed at the machine language level. Since each type of CPU has a different instruction set, uses different mnemonics to identify instructions in the set, and uses different binary patterns to identify each directive to the CPU, it means people desiring to accomplish the same task using two different types of CPU devices will have to do almost twice the work of that required had they used the same CPU. The development of the actual algorithm to accomplish the desired objective would normally apply to both systems. However, the actual implementation of the algorithm will require separate development paths in order to specify the actual machine language instructions that are to be used in each machine. Each type of instruction must be identified with its mnemonic. Then the mnemonics must be converted to completely unrelated binary patterns. (Notice that the problems just mentioned which are present at the machine language level disappear when one utilizes a standardized high level language to solve an identical problem on two different machines!)

The amount of extra work required to implement similar capabilities on two different machines when working at the fundamental machine language level is a prime reason why an individual microcomputer system owner-to-be should give considerable attention to the type of CPU device on which the system will be based! If large groups of people are all using computer systems based upon the same type of CPU, a considerable amount of common benefits can accrue through the mutual exchange and development of machine language programs. The more types of programs available, the more potentially useful the machine becomes! Out of the numerous different microprocessor CPUs available on the market, most personal computer systems are based on a few of the most popular types. A prospective

system purchaser should become acquainted with those considered most popular by reading current trade journals at the time one is considering acquiring a system. However, since the popular favorite will undoubtably change with time as technology advances and manufacturers promote new CPU capabilities, the person with a system that allows the CPU module to be replaced with a different type will probably find it a worthwhile feature. This is particularly so if one plans to build up a sizable system over a period of years. (Though, at the rate technology is moving, and the rate at which people will probably start applying personal computer systems to dedicated applications, this strategy contains no guarantee that one won't find it desirable or advantageous to purchase complete new systems in the future, rather than just upgrading a CPU card!) In any event, the importance of being able to swap different types of CPU modules in a computer system should be considered a luxury in relation to the importance of being able to plug in various types of memory modules into a computer!

A computer without memory elements is like a railroad locomotive without any cars to pull. It cannot carry any fuel (instructions) to keep itself running, and it has no place to carry cargo (hold data). Like a locomotive, the more memory (cars) a computer system contains, the more instructions (fuel) it can store, and the more data (cargo) it can handle "on-line" (without having to revert to the use of relatively slow I/O devices). A computer with a backplane board allows modules of memory to be plugged into the system to provide greater capability just as one might couple extra cars onto a locomotive to enable it to pull more cargo. Many microprocessor systems allow modules of memory to be added to a system in multiples of, for instance, 1024 bytes up to the maximum addressing capability of the CPU. (Maximum addressing capability is typically 65,536 bytes in current machines. That is the maximum number of addresses that can be specified in a 16-bit binary register. The reader may recall that 16 bits is the commonly provided length of microprocessor program counter registers.)

The prospective computer user should know that there are several types of memory elements which may be installed in a computer. Each type has specific characteristics which provide benefits and drawbacks with which one should be acquainted.

The most common type of memory elements used are semiconductor RAM devices which were mentioned previously in this publication. These devices can have their contents altered under the direction of the CPU and are referred to as "read and write" memory elements. The abbreviation RAM, the reader may recall, actually stands for "random-access-memory." This means the elements may be almost instantaneously accessed by specifying an address, without having to go through any special sequence of addressing operations as would be the case with so-called "serially-accessed" memory elements. A drawback of semiconductor RAM elements is that when power is removed from them, their contents are lost. When power is initially applied to them, the individual cells making up the memory elements will be in essentially random states. Hence, they must be "initialized" by loading them with information each time power is applied before they can serve any practical purpose.

There is another type of random access read and write memory. It has been in use for many years in larger computers. This memory is not made out of semiconductor material, and is not manufactured as an integrated circuit. It is made out of magnetic material. The material is shaped in the form of a microscopic doughnut. Each doughnut is but a few millimeters in diameter. One such doughnut is required for each bit of memory. Extremely thin wires are passed through the center of the doughnuts. When the proper signals are passed through the wires, individual doughnuts may be placed in one of two possible magnetized states. Each state represents a logic level. When no signals are passed through the wires, the individual doughnuts retain their most recent magnetized state. Other thin wires passing through the centers of the magnetized miniature rings are used to determine what state a particular ring (representing a bit) is in. The doughnut shaped rings of magnetic material are often referred to as CORES, and a bank of memory elements made up of such CORES is generally referred to as a CORE MEMORY.

So-called core memory elements are very seldom provided in microprocessor based computer systems for a number of technical considerations. Core memory banks are relatively expensive compared to semiconductor devices when configured for small systems. They require the support of relatively sophisticated electronic circuits which are more difficult to maintain and service than semiconductor

memory chips. They require more space and hence, individual modules cannot hold as many bits of information as presently available integrated circuit modules. They often require special power supply voltages. Having to provide these special voltages can mean added expense in a microcomputer system. There are other drawbacks that prevent them from being used to any great extent in microprocessor systems.

If they have all those drawbacks and are seldom used, why are they even mentioned? Because they do have one highly useful feature that in certain critical applications can be of paramount importance in a computer system! When power is removed, a core memory bank does not lose the information stored in the magnetic fields of the cores! When power is applied to a system with core memory, the computer can immediately commence operating using the contents of the core memory to obtain instructions or data because the instructions or data will be in just the state they were left in when power was removed from the unit. This kind of capability is sometimes required in, for instance, critical industrial process control applications. In such situations, a temporary power failure that might put a semiconductor memory equipped computer out of business by destroying the contents of the system's memory banks, would have no such devestating effect on a core equipped microprocessor controller.

Most people using a microprocessor system for personal use will have no requirement for core memory. The reader will soon learn that there are special memory elements now available for use in computers that provide the capability to quickly bring a computer "back to life" after a power failure. However, the reader might be interested in knowing that prior to the development of semiconductor memory integrated circuits, core memory elements were used for the memory banks of the vast majority of computers. The reader should always be able to spot an "old-timer" in the computer field by observing the reaction of that person to the mention of "core memories." If there is no reaction - the person probably hasn't been around computers all that long! For the reader's own sake, references to "core" in computer literature are still frequently encountered. It is still sometimes used as a simple synonym for "memory."

There is a type of memory element frequently used in microcomputer systems that has one advantage over the semiconductor read and write (RAM) memory elements previously discussed. The advantage is that when power is removed from the devices, their contents are not altered. These devices also have a major disadvantage. They cannot have their contents altered even when power is supplied! That is, the computer cannot "write" new patterns into memory addresses which are occupied by these types of elements. These types of elements are called, simply enough, "read-only-memories." Read-only-memories are generally supplied in the form of integrated circuit packages. They are abbreviated and often referred to as ROMs.

At first introduction to the idea of a memory that can only be read from and not altered or written into, some people think the concept is some kind of joke! "What good," they ask, "is a memory that cannot be altered by the computer?" Well, it certainly isn't any good for storing temporary data, such as the results of calculations! But, it certainly is a fine element for storing important, frequently used computer algorithms or programs, on a permanent basis! Why do that? Remember that semiconductor RAM memory elements lose their contents whenever power is removed from the system in which they reside. Accordingly, whenever power is initially applied to a system, the RAM elements will be in a state of complete disarray. This means, in a system containing only RAM elements, that when a computer system is initially powered-up, it is completely useless! The memory elements will not contain any type of a program directing the CPU to perform any sort of useful operations. What a mess! A nice, potentially powerful computer system turns out to be virtually useless until at least a portion of its memory cells can be put in states (filled with binary patterns) that can be properly decoded by the CPU in order that it be able to accomplish a useful purpose!

How does one get around this problem which has plagued computer users since the time the machine was invented? Well, many computers are provided with special switches on their front panels that enable an operator to manually "toggle-in" instructions into locations in memory. Using this method, one can initially load a program into memory to get the computer started. Usually the initial program that is manually "toggled" into memory using front panel switches is some sort of minimal series of instructions that will allow

the computer to use a bulk storage I/O device. Quite often, the first program brought in via such a device is a more powerful, general purpose memory loading program. The general purpose loading program is then used to bring into memory whatever full-sized program or data one plans to have the computer utilize.

The very first "minimal" program that one toggles into a computer's memory banks in order to get a system operational is often referred to as a BOOTSTRAP LOADER program. This nomenclature comes from the old expression "to lift oneself up by one's own bootstraps." In effect, that is what the computer does in the process just described. First, an operator uses special front panel switches to place a short program into memory and to direct the computer to commence executing that program. The computer starts "using itself" from then on by executing that short program to operate an input device that will read in a longer program. The first program brought in from the input device is often simply a more powerful loading program than the initial "bootstrap" algorithm that was toggled into memory. This is because the toggled in program is typically designed to utilize a minimum amount of actual instructions - to shorten the manual labor of toggling in the instructions - and may thus not be a highly efficient input device handler. So, in effect, the initial bootstrap program is often used to read in a program that is really just a more powerful bootstrap program. That program then serves as the final bulk storage handler. The computer thus uses its own capabilities to increase its own power by successively loading itself with increasingly powerful programs. It pulls itself up by its own bootstraps! A pretty neat concept if one stops to think about it!

From an operator's viewpoint, the whole bottleneck in bringing a computer "on-line" lies in the initial process of having to manually load in directives into the computer's memory using toggle switches. Wouldn't it be nice for the operator if, whenever one applied power to a computer, there was some section in memory that did not come up in completely random states? Wouldn't it be nice if, whenever power was initially applied to a microcomputer, a section of memory contained a program? A program such as a bootstrap loader that the computer could immediately execute, so that an operator would not have to spend several minutes keying in such a program? Yes, indeed it would! And that is exactly what a read-only-memory is capable of

providing. Whenever power is applied to a system containing ROM elements, those elements will contain patterns (predetermined by the ROM manufacturer) that can serve as instructions (or data) for the CPU.

When ROM devices were originally developed, they were relatively expensive devices. It was considered a luxury to have a 32 byte ROM in a computer in which one could store a rudimentary bootstrap loader program. However, in recent years the prices of ROM elements have fallen dramatically. If a particular pattern is encoded in a ROM device in sufficient volume, the price approaches and even goes below the price of read and write elements. Thus, today one can obtain ROM devices for microcomputer systems that contain several thousands of bytes of instructions. Such ROMs can hold entire Monitor programs giving instant useful capability to a microcomputer as soon as power is applied. Going one step further, some computer systems can be equipped with a series of ROM elements programmed to contain entire high level language interpreters. As soon as power is applied to such a ROM equipped computer, one may commence to program the machine using a high level language such as, for instance, BASIC. Admittedly, this kind of capability, while being aimed at by increasing numbers of computer system manufacturers, is still somewhat of a luxury. In most typical personal computer systems today, the use of ROM elements is generally limited to providing fundamental bootstrap, or general Monitor program capability.

In some applications, such as machine controllers, a computer might have all its memory dedicated to ROM elements. Such usage, however, would severely restrict the amount of data that could be inputted and manipulated by the computer at any one time since all such temporary information would have to be held in CPU data registers. In most cases, a system using ROM elements will also contain read and write elements. The RAM elements can then be used to hold temporary data being inputted to and processed by the computer, or for holding other programs that might be loaded into the computer by bulk storage I/O devices. In general purpose personal computer systems, the amount of read and write memory typically far exceeds the amount of ROM memory.

It is important to realize too, that the patterns placed in ROM elements are instilled there by the manufacturer during an expensive process. This process only becomes economically practical when the volume of identically patterned units is quite large. Thus, ROM elements containing sequences of instructions are only produced for applications that are in considerable demand, and for machines that use highly popular CPU devices. This is done so as to retain as wide a market base as possible. The microprocessor owner who has a unique algorithm that it is desired to have available in permanent form in a computer system does not have a ROM manufactured. That person can, however, use a special type of read-only-memory element called a PROGRAMMABLE READ-ONLY-MEMORY (abbreviated PROM).

A PROM is similar in concept to a ROM in that once a pattern has been established in the device, it is non-destructible under normal use (meaning it cannot be over-written by the computer). It also retains its pattern if power is removed. However, instead of a pattern being established by the manufacturer, a PROM is initially produced so that all of the bits within it are in one "unprogrammed" state. That is, depending on the type, all of the bits within a PROM will all be in a logic one or logic zero state. Using special circuitry, often provided as a special module that can be driven by a typical microcomputer unit, a PROM device may be programmed by the user ONE - AND ONLY ONE - time! Once the user has "programmed" a PROM with a pattern, it retains that pattern permanently.

PROM devices are nice when an individual has a unique program or set of algorithms that it is desired to have available as soon as a computer system is powered-up. The devices typically cost three or four times what a similarly sized ROM (size referring to the number of bits or bytes on the device) might cost. However, they provide a reasonable compromise between having to "go without" when a particular series of instructions (or data) may not be of interest to a sufficient number of users to tempt a manufacturer into producing a ROM. The reader should be cautioned that adding to the price of PROM devices themselves is the cost of the programming equipment necessary to establish the desired patterns in the devices. Such equipment can cost several hundred dollars or more. Often, groups of computer enthusiasts join together to finance the purchase of such a unit

which is then shared. Also, a few commercial firms that have the required equipment will program PROM devices to customer specifications for a modest fee.

There is another type of read-only-memory device that has become quite popular with small system owners. It is called an "Eraseable Programmable Read-Only-Memory." What this means is that, while the elements may be considered as "read-only" when installed in a microcomputer, they can actually be erased and then "reprogrammed" with a different set of patterns using special procedures. Typically the procedure involves exposing the memory elements to high intensity ultraviolet light. This process restores the elements to their initial all zeros or ones state from which they can be reprogrammed using special programming equipment. Frequently, the equipment necessary to program the EPROM devices is of the same nature as that used to program PROMs. Hence, for the added investment of a special ultraviolet light, for instance, one may use read-only-memory elements that may be recycled. Since EPROM devices have roughly the same prices as PROM devices which can never be reprogrammed, it is easy to understand how EPROM units have become popular among personal computer enthusiasts. In fact, they are highly popular with commercial users who like the extra protection of being able to recover read-only-memory devices. After all, even the professionals make programming mistakes or change their minds about the kinds of programs they want "permanently" installed in a computer.

PROM and EPROM devices are commonly available as integrated circuit packages containing, typically, 256, 512, or 1,024 bytes of memory. The costs (unprogrammed) run from $10 to $15 for the 256 byte units on up to $100 or so for the 1,024 byte packages. If custom-programmed by a commercial outfit, the cost can double. This is because the patterns to be programmed on the devices must be inputted to a programming machine. If this is done manually, the price is high. If semi-automatic methods are used, such as reading the patterns from a punched paper tape supplied by the customer, the price for placing patterns on a PROM is usually quite modest.

ROMs, PROMs, and/or EPROMs are frequently plugged into printed circuit boards so as to form a module. Such modules typically hold a number of the devices so that they can store anywhere

from 1,024 on up to 8,192 bytes, depending on the types and number of integrated circuits installed. Typically, a block of memory addresses in a computer system will be reserved or a suggestion made that an area be set aside for the use of read-only-memery elements. An important item to remember when using such elements is that when the patterns encoded represent instructions, the elements will normally have to be addressed at specific locations in memory and cannot be moved about. This is because certain classes of microprocessor instructions in most machines are address specific (such as JUMP and CALL instructions). That is, bytes contained in the instructions identify specific memory addresses. If a ROM device contains instructions of that type, which refer to specific locations within its own address boundaries, it cannot be relocated to another area in memory. This is because, when relocated, the specified addresses would not be at their intended locations. I.e., the locations of instructions referenced by JUMP and CALL directives would have changed.

ROMs are sometimes used to hold tables of data which might be frequently referred to in a computer system. This is often done if a particular application calls for using conversion tables, or in certain mathematical applications, such as determining the sines and cosines of angles using table lookup methods.

In summary, read-only-memory devices are useful whenever it is necessary to have instructions or data instantly available in a computer system as soon as power is applied. Remember though, that they have a major drawback: they cannot be modified or written into when installed in a typical microprocessor system. ROM devices can never be altered. EPROM devices must be cycled through a relatively lengthy process requiring removal from the computer, ultraviolet light erasure, and reprogramming by special equipment. (Remember that RAM memory, the mainstay elements in typical general purpose systems, can have their contents altered under control of the CPU in but a few millionths of a second!)

O.K., the reader now knows that two classes of modules typically provided in microcomputer systems are CPU modules and memory modules. A person selects the type of CPU that one desires to use and purchases or builds a module that will contain or support that

type of central processor unit. Except in very special cases beyond the scope of this publication, there will only be one CPU module in a computer system at any one time. However, as has been pointed out, one may want to consider obtaining a system that will allow different kinds of CPU modules to be plugged into a system when different kinds of capabilities are desired. This capability is also valuable as a hedge against a CPU becoming outdated by technology. On the other hand, a typical general purpose microcomputer system may contain a number of memory modules at any one time. Memory modules may be added to a system to increase the overall memory size as funds become available for such purposes, or as the need arises for such capability. (Generally the computer enthusiast will find the need or desire for the extra capability is always one step ahead of the funds!) Also, the kinds of memory modules in a system may be mixed among RAM, ROM, and PROM or EPROM devices. In fact, when planning a system, it is a good idea to check and see that the manufacturer or designer made provisions for allowing a goodly number of such memory modules to be added to a system. Preferably, the number allowable is such that the memory banks may be expanded up to the maximum addressing capability of the CPU that one plans to utilize.

A modular system will also have provisions for plugging in a variety of I/O INTERFACES. These are printed circuit boards or cards containing the circuitry necessary to pass information to/from the computer and external devices such as keyboards, printers, bulk storage devices, and so forth. For instance, if one planned on using an audio cassette recorder to make copies of programs and data stored in the computer's memory, as discussed in the previous chapter, one would need a special interface module. That module would have to be capable of taking digital signals from the computer and converting them to audio tones to be recorded on the tape. It would need to be able to decode special patterns received from the computer so that it could determine when to perform (start and stop) such operations. It would need to provide control signals to the audio tape recorder machine. It would also need to be able to perform the reverse procedure of converting audio signals received from the tape unit when in the playback mode, back to digital signals in order to transfer the recorded material back into the computer. It would also need to generate control signals when operating in the playback mode to in-

form the computer whenever a new character was ready to be inputted, etc. Such an interface would typically be called an AUDIO CASSETTE INTERFACE or MAGNETIC TAPE INTERFACE.

In a modular system, one will typically need to have an I/O interface module for each kind of device (and, in principle, for each device, even if one has several of the same kind) connected in a system. Thus, if one plans to have an electronic keyboard, an electro-mechanical printer, and an audio cassette recorder connected in a particular system, one will need to have a specific keyboard interface, a printer interface, and an audio cassette recorder interface. Three different interfaces for the three different types of devices.

Naturally, a system that allows various types of I/O interfaces to be plugged into a backplane allows for flexibility. One will find, if one gets involved in the field, that the desire for flexibility in interfacing various types of devices to a computer is quite strong. This is particularly true as different types of I/O devices become available (through development) that allow a computer to be connected to more and more kinds of devices and instruments.

Some I/O modules combine several different types of interfaces on the same unit. Thus, for instance, one may find a manufacturer that has all the circuitry for driving a TV display as well as accepting inputs from an electronic keyboard along with, say, an audio cassette interface, all on one module. This technique can often reduce the overall cost of connecting to a group of specific devices since it eliminates having three separate modules. It can also increase the number of devices that may be connected to a computer at one time by leaving more connectors open on the system's backplane for other interfaces. Sometimes the technique of combining interfaces onto one module may be adopted as an inducement to use a particular type of I/O equipment supported by the computer manufacturer. In actuality, all interfaces must be designed for specific types, or at least specific classes of equipment. The prospective system user should be sure and obtain technical information relating to any I/O interfaces one plans to obtain, in order to ascertain that they will serve the actual I/O devices one intends to use.

There is one other type of module that is provided in some micro-

computer systems. This module might be called a FRONT PANEL or MANUAL CONTROL module. This type of module provides capabilities for an operator to control a computer system through the use of switches and visual indicators. Such a module typically provides capability for an operator to access specific locations (addresses) in memory, and to alter the contents of those locations addresses (in order to load programs or data). Frequently, a FRONT PANEL module also allows an operator to direct the microcomputer to perform specific individual instructions. The module also allows the computer to be placed into certain modes of operation such as: The automatic program execution or RUN mode. A SINGLE-STEP mode whereby the computer will execute a series of instructions, one instruction at a time, each time a "step" switch is activated. A HALT mode to stop the operation of the microcomputer. Other modes that provide similar types of controlled operation.

Quite often the FRONT PANEL or MANUAL CONTROL modules are designed to use ordinary toggle switches for inputting of information. A simple arrangement of lights is then used to display the status of bits (such as the conditions on address, data, or control busses; or the conditions of individual CPU registers such as accumulators, memory pointers, and so forth).

The next step up from a simple arrangement using toggle switches and lamps is a configuration consisting of an octal or hexadecimal keypad and groups of numeric or alphanumeric displays that can present the same type of status information in a more compact form. Naturally, the latter arrangement is somewhat more convenient to use. The cost of the latter type of module is also somewhat more than the former.

Front panel or manual control modules are necessary to a varying degree depending upon the extent to which ROM or similar read-only-memory elements are utilized or provided in a system. Remember, a computer that does not have any read-only-memory elements installed or similar non-volatile elements that will retain their contents when power is removed (such as core memory units) must have some method whereby the computer can be manually controlled. This is necessary if only for the purpose of being able to load a minimum bootstrap loader program that will in turn allow external equip-

ment to load the computer's memory banks. Such a method is faster than that of a human toggling in programs or data on a byte-by-byte basis. On the other hand, if a system contains ROM elements, the requirement for manual controls as typically provided on a front panel is minimized. The ROM elements can hold programs that will allow a more sophisticated input device, such as a keyboard, to be used to issue commands to the computer through a program such as a Monitor package. Many microcomputers available today do not have manual control modules. Instead, simple circuitry insures that the CPU's program counter will be initialized to the start of a Monitor program (which resides on ROMs) as soon as power is applied to the system. This feature immediately enables a computer user to communicate with the system using a keyboard and suitable display device. Or, in a microcomputer dedicated to a particular application, such as controlling machinery, initial application of power will automatically result in program operation being started at an address dedicated to ROM elements that contain appropriate instructions. As a general rule, the more sophisticated a system is in terms of the operating SOFTWARE (programs) supplied on read-only-memory elements in the system, the less the requirements are for manual controls. Of course, some systems provide both features - a fully equipped manual control facility and a good Monitor program on ROM elements. However, the reader should bear in mind that both features have their price, and a goodly number of system owners have elected to do without one or the other. The trend is definitely away from front panel switches and indicators. It is towards increasing the capability of software supplied on read-only-memory elements. This trend may be expected to continue in the future. In fact, the general availability of essential operating programs such as Monitors on ROM elements for the most popular types of microcomputers has pretty much eliminated the need for heavily switch and light-laden front panels of earlier days. There is little doubt that some manufacturers provide (and charge for the feature) traditional "flashing-lights" types of front panels more for cosmetic reasons than because of their practical functional capability. The reader that plans on acquiring a computer and doing more serious things than just watching blinking lights might give careful consideration to doing without a fancy front panel or manual control module. Provided, of course, that the system has the capability for being appropriately controlled through more convenient devices using a Monitor program stored on ROM elements.

Regardless of whether or not a system includes a front panel or manual control module, a user of a modular system must keep in mind several parameters when selecting or expanding a system of modules. One parameter is called BUSS LOADING. This parameter has to do with the number of modules or circuits that one may safely connect to the signal lines that form the backplane buss network. These buss lines are limited by electronic considerations to supporting some maximum number of connections. This comes about from the fact that modules that must supply signals to the buss lines are designed to output signals of sufficient strength to properly drive some finite number of modules. They cannot supply or properly operate an unlimited number of other modules. The number of modules that may be interconnected within a particular system may vary greatly depending on the system manufacturer. However, there will be some maximum number. The user should make sure that the number is known and adhered to when assembling a system!

Sometimes, the number of other modules or circuits that a module is able to feed a signal to (drive) is referred to as the FAN-OUT capability. The FAN-OUT capability is usually referenced to a standard electronic "load" figure related to current and voltage. A "fan-out" figure of ten, for instance, would mean that the device could drive ten standard "loads."

As modules that provide signals have so-called fan-out capability, modules that accept signals, and thus present a "load" to a driver, are often given a FAN-IN figure of merit. Without getting things overly complicated, the FAN-IN figure refers to the number of driving circuits that may be connected to an input line simultaneously. In summary, neither the fan-in or fan-out capability of a module should be exceeded in a system. (Normally, in a typical computer system, one is more likely to be concerned with the fan-out figure.) Manufacturers generally state the fan-out or fan-in "loading" figures for various modules they produce. In most state-of-the-art systems, one will run out of connectors on a backplane due to physical size limitations, etc., before one has to become concerned with exceeding module driving capabilities. However, one should be careful when mixing modules made by different manufacturers in the same system. One producer may not follow the same design considerations as

another when it comes to providing drive capability, fan-in loading, etc.

Also, the prospective purchaser of a very small "introductory" or beginners microcomputer system should beware of the parameter being discussed. Systems designed to be expanded to any great extent using modules generally have special circuits called BUFFERS, BUSS DRIVERS, or LINE DRIVERS installed on modules in order to support the addition of a large number of modules as the system is expanded. Many "tutorial," "training," "educational," "experimental," or "what-have-you" microcomputer systems do not - repeat NOT - have the appropriate buffers or drivers provided on the basic units. Hence, unwary purchasers who acquire such units, and later desire to greatly expand their systems, may find themselves in a situation where a system cannot be readily expanded unless they have the technical expertise to design and build special buffers and expansion logic themselves. The point is, investigate first! Check to see if a small system can be expanded, and if so, just what is required for expansion to various stages.

Another parameter that prospective system owners should pay close attention to has to do with power requirements. Computers require electric power in order to operate. This power is provided in the form of current supplied at specified voltage(s). Generally, a special POWER SUPPLY must be provided to convert ordinary house voltage with its alternating current to a relatively low voltage level that supplies direct current. Some computer systems require several levels of low voltage to supply the necessary power. Others need just a single voltage level. Most often, the single supply level required is five volts.

Low cost microcomputer systems are generally designed so that the POWER SUPPLY for the system is a completely separate unit. The supply may be purchased separately either from the computer manufacturer or other outlet. (Some microcomputer suppliers do not produce or otherwise make available such units.) The power supply must be connected to the microcomputer backplane or appropriate module by a power cable. Larger, more expensive microcomputer systems usually have provisions for a power supply to be included in the system's main cabinet, or it is designed as an integral part of the

computer system. The point being made here is that the reader purchasing a system should ascertain whether the unit is provided with a power supply, or whether it is considered a separate item!

Once that has been established, there is still more to be ascertained by the prudent shopper. Just how much does the computer system one plans to initially start with, as well as ultimately expand to, require in terms of electrical power that is to be provided by the system's power supply? This information, once calculated, may be used to determine whether a power supply provided with a system will be sufficient to meet present and future needs. Often it is possible to obtain power supplies of differing capabilities and different costs. It is not economically attractive to buy a large, expensive power supply if one will never utilize anywhere near its capabilities. On the other hand, it can be just as wasteful in the long run to obtain a small power supply for an initial small system which one has definite plans to expand, when doing so will result in one having to eventually purchase a more expensive, larger unit at some future date. If the smaller unit can be upgraded or expanded (not usually the case with power supplies), fine! If it must be replaced (meaning the smaller supply becomes obsolete or useless), then it is not so fine.

Generally, in a modular system, each type of module that may be installed in a system will have its own individual power requirements specified in terms of voltage and current. (If multiple voltages are required, the current for each voltage level needs to be indicated in order to determine power requirements.) Current requirements are specified in amperes or milliamperes (thousandths of an ampere). Typical modules might require anywhere from 250 milliamperes (0.25 ampere) to 1.5 to 2.0 amperes (1500 to 2000 milliamperes) at, for instance, a level of 5 volts. Power in watts is simply the product of current in amperes times voltage. From this information, the person developing the system can determine the total power requirements for a proposed system. This can be done for both the initial configuration, which may be quite small, and a maximum or ultimate configuration that one might plan (dream?) of expanding to over a period of time. One needs simply add up the current requirements for each type of module in the system at each voltage level required. The sum gives the required capabilities of the power supply at each voltage. Generally, it is a good idea to add a little to the maximum

anticipated figure for safety, or possible unanticipated requirements.

A modular system is assembled by essentially plugging appropriate types of modules, such as the types described, into a backplane arrangement. Then, one must hook up an appropriate power supply if it is not already an integral part of the system. Of course, cables must be used to connect from I/O interface modules to actual I/O devices unless the devices are included as part of an overall computer console. (In the beginning days of personal computer systems, I/O devices were almost always considered as items separate from the computer. The current trend, however, is towards including as much as possible, in the matter of such devices, as part of a computer console. The practice is certainly convenient from an operator standpoint. Such design can also reduce the number and/or length of cumbersome interconnection cables if the appropriate signal lines can be provided as part of a backplane or printed circuit wiring arrangement in the console.)

In practice, a computer system generally contains hundreds to thousands of physical circuit connections. These circuit connections are accomplished by soldering, using printed circuit connectors, point-to-point wiring, cable connectors, and so forth. Integrated circuits, for instance, must have each of their pins soldered to the printed circuit foil on a card. Or, they must be inserted into integrated circuit (IC) sockets with the sockets in turn being soldered to printed circuit cards or otherwise interconnected with other sockets containing integrated circuits. Individual modules are similarly interconnected by their being plugged into card sockets whose pins are soldered to backplane busses or otherwise interconnected. Cables, often joined by cable connectors, are frequently used to connect power supplies and PERIPHERAL (auxiliary) devices such as I/O equipment. Each physical connection in a computer system is a potential weak link. The number of weak links in even a medium sized computer system can become quite large. For instance, the memory banks alone in a medium sized microcomputer system having, say, about 16,000 bytes of memory, may be made up of 128 integrated circuits. Each IC might have 16 pins which must be interconnected to form the memory banks. One may calculate that there are close to 2,000 physical connections involved in the memory banks section of such a system alone!

O.K., so there are a lot of connections in a computer system. Why is all of this being mentioned? To point out to the reader that while a microcomputer system is conceptually not all that difficult to understand, it is by no means a particularly simple piece of equipment in terms of HARDWARE (the physical elements of which it is made)! If just one connection in a computer is not made properly, or one part becomes defective, a computer can malfunction. While, with the proper training, a person can learn to troubleshoot and repair a computer that is not operating properly; without the proper background, a person can be pretty much at a loss when it comes to repairing a malfunctioning computer. This fact is a consideration a prospective computer owner should consider when one is deciding what kind, and in what form, one might purchase a microcomputer system.

If one is technically competent in electronics (has training as an electronic technician for example) and is into the pastime of building electronic equipment, one can indeed build a microcomputer system from "scratch." One can, for instance, obtain plans for constructing various types of system from CPU manufacturers, or in trade magazines and journals. The same goes for building peripheral devices such as CRT displays, and for constructing appropriate I/O interfaces. Using this method, one etches their own printed circuit cards, or uses point-to-point wiring methods on special types of boards designed for electronic wiring work. One obtains the necessary components, integrated circuits, resistors, capacitors, transistors, and so forth from various suppliers such as distributors and parts stores. Following construction plans such as board layouts and parts placement diagrams, and using schematic diagrams, the builder assembles the various components and interconnects them. Finally, the typical builder can expect to spend some time after the unit is physically constructed performing operational tests to make sure everything is working properly. Often at this stage, it will be necessary to use electronic troubleshooting methods to locate defective components, improper or missing connections between components, etc. The process is not for those who are not properly trained and skilled. However, some people with the appropriate backgrounds thoroughly enjoy the aspects of building and checking out a microcomputer, as they typically enjoy other types of electronic construction projects. Readers who have such capability are certainly encouraged to go that route if they have the ambition. That kind of familiarity with a system is

worth a considerable amount when it comes to maintaining and servicing a system over the long run.

Readers that do not have the background to tackle building a computer from the ground up, but who have some training in electronics (perhaps from experience as a serious enthusiast) may want to consider the possibility of constructing a computer from a kit.

There are numerous microcomputer manufacturers and suppliers that put out packages of components and parts, along with instructions for assembling the parts together to form computer systems. However, the reader is cautioned to evaluate various suppliers carefully in several areas. The amount of "instructional material," assistance, documentation, and the extent of guarantees and warrantees varies considerably among suppliers. In fact, when it comes to computers, the factors mentioned may well be of more importance for the individual purchaser than the factor many people tend to immediately associate with: the quality of components used. Because of the nature of the machine (the fact that it is primarily made up of various types of integrated circuits that are either functionally useful or not), the quality of the actual operational parts from supplier to supplier is not as likely to vary as is the quality of supporting services of the type mentioned. (Of course, there are exceptions!)

It is a good idea to get samples of the documentation supplied by various kit producers in order to determine which ones provide packages at a level suited to an individual's experience level. Some kit suppliers provide construction information at a level suited for electronic technicians who understand the basics of electronics construction. Only a few produce packages suitable for the rank beginner. In addition to reviewing the actual construction manuals, one should also carefully check how much information is given regarding testing and troubleshooting of equipment that might not function correctly after it has been built. Finally, check for a good guarantee or warranty and find out about service arrangements beyond that period. With modern technology, once a computer is operational, it may run several years or more without problems. However, if a problem does develop down the road, it's a good idea to know beforehand just where service can be obtained, and what it is likely to cost.

Building a computer from a kit can save as much as 20 to 25 percent in some cases over purchasing a similar system assembled and tested. However, if a person has no suitable background in electronics, or does not desire to become involved at the components level, there are many microcomputer system producers (including many of the firms that provide kit versions) that sell systems fully assembled and tested. The number of such manufacturers is increasing rapidly. As with kit suppliers, it's a good idea for the beginner to shop around. Look at literature. Talk with other people who may have acquired various brands. Find out about service, warrantees, and other types of support. Are maintenance and technical manuals available with the equipment?

Regardless of how one intends to actually assemble a system, whether by building it from scratch, constructing it from a kit, or obtaining a unit already assembled and tested; there are a number of matters one may want to consider relating to the ultimate size and use of the system being planned. The following discussion will point out some of the ramifications of choices based on these factors which the reader may wish to consider.

For purposes of discussion, three types of microcomputer systems will be considered: A small system primarily suited for instructional, experimental, or machine control purposes. A medium-sized system suitable for many "personal" applications. (Such a system would be capable of performing a wide variety of services, including supporting advanced machine language programming development aids and/or a high level language). Finally, some aspects of a large microcomputer system designed primarily for small business applications will be discussed. Certain parameters are more important, typically, in each type of system, as will be discussed.

A person interested in obtaining a small microprocessor system for purposes of education or to experiment with, or a person desiring to use a microcomputer as a controller, can get by with a minimum system. A number of outfits produce and sell such systems. In many instances, they stress them in their advertisements as being designed as "product evaluation" units. An individual can have a lot of fun with these units, and apply them admirably to a great number of tasks, provided that they make sure the equipment comes with, or

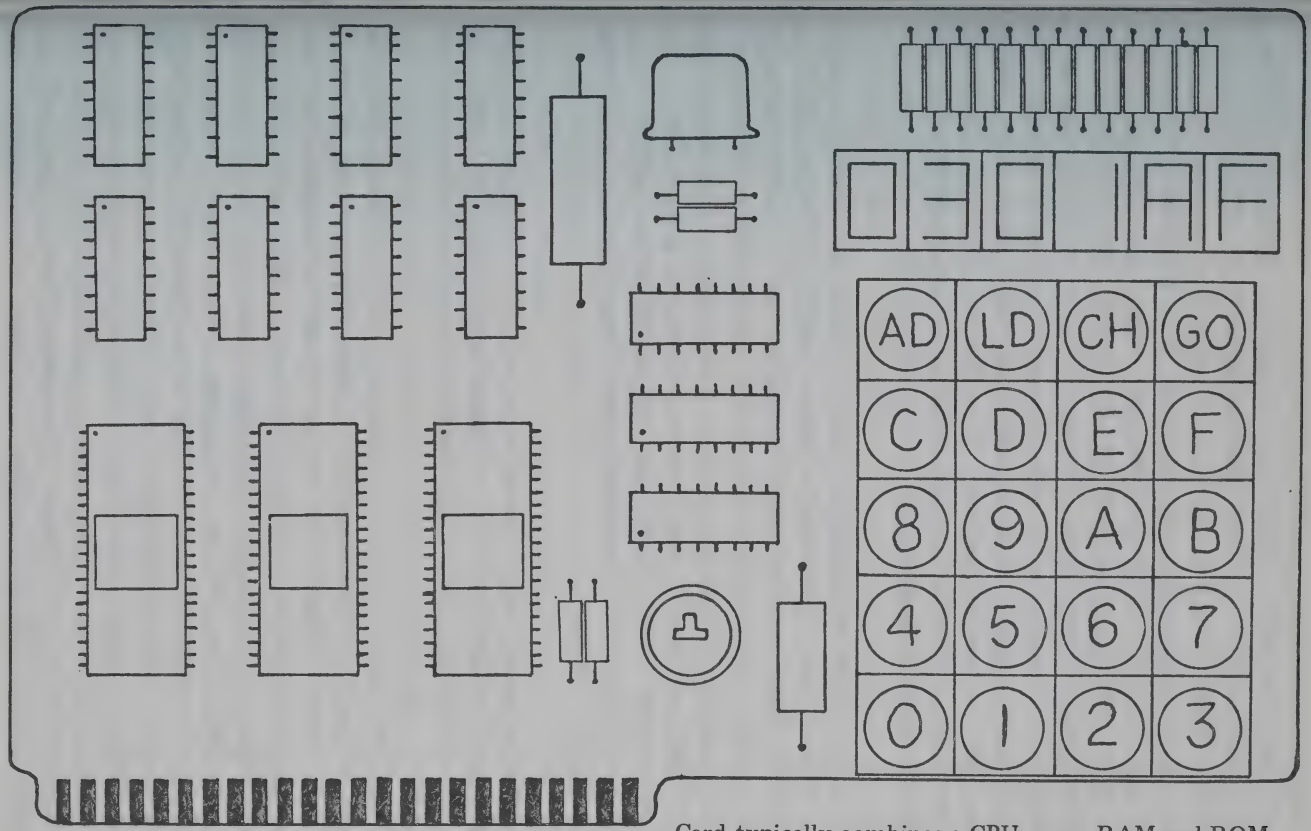
can be supplied or connected to, several types of I/O devices. These I/O devices can make all the difference between the system being practical and useful, or essentially useless once one has exhausted the initial educational or evaluation purposes of the unit.

The types of I/O devices that it is essential to make sure that a minimum microcomputer system can work with are: A convenient input device such as a keypad or keyboard (versus toggle switches). A simple display (output) device such as seven-segment display lights that can directly display register contents in octal or hexadecimal format (versus binary lights). And, last but not least, some sort of bulk storage interface, such as an audio magnetic tape unit, should be provided or be readily available so that one may have a convenient means of storing and retrieving programs!

It is also advisable to make sure that such a system comes equipped with FIRMWARE (software supplied on a read-only-memory) that will support the indicated peripherals. At the very least, a ROM or PROM containing a bootstrap loader that can read in a Monitor using a bulk storage device should be available.

The individual obtaining such a unit for private use should watch out for evaluation units intended primarily for use by electronic engineers in businesses. Such evaluation packages are often designed to interface with, and have software designed to support, relatively expensive electro-mechanical keyboard/printer combinations, particularly those models that have paper tape reader and punch capability as auxiliary features. In such cases, it is possible to obtain a perfectly fine small microcomputer system that sells in the $100 to $200 range only to find that to effectively utilize the firmware supplied with it, one must connect the unit to an I/O device that can cost $1000 or more. That may be satisfactory for a company or organization that normally has such I/O equipment around as a normal course of business. It is not so nice for an individual that does not. Particularly if the discovery for the requirement for the expensive I/O device is made after one has laid out a couple of hundred bucks for the microcomputer!

Fortunately for individual enthusiasts, a number of microcomputer suppliers have realized there is a sizeable market serving individuals



TYPICAL "EDUCATIONAL" MICROCOMPUTER

Card typically combines a CPU, some RAM and ROM memory, a keypad, display, and an audio tape interface. Edge connector allows limited expansion.

and firms who may not have expensive electro-mechanical keyboard/printing units. Or who consider it a mismatch of funds and capabilities to couple such a unit to a tiny microcomputer system. These suppliers typically provide complete miniature microcomputer systems that contain on a single printed circuit card: A CPU, memory (typically consisting of from 128 to 1024 bytes of RAM, and perhaps twice as much ROM containing system operating routines), a simple octal or hexadecimal keypad for inputting information to the system, a small display consisting of six (hexadecimal notation) to nine (octal notation) seven-segment display devices, and often the circuitry necessary to convert digital data to audio tones for recording on a tape recorder, and circuits to perform the reverse operation to recover data from a tape recorder (an audio tape interface).

The prospective purchaser should be aware that many of these tiny systems do not come with a power supply, or the power supply must be purchased separately at additional cost. When a power supply is not supplied, and not available from the computer supplier as a separate item, the buyer should check carefully to determine what type (voltage and current requirements) of power supply is necessary to run the unit. If the system has expansion potential, the reader should consider future power needs when obtaining the power supply from a suitable source.

The reader should also beware of the fact that some evaluation systems are not designed to be extensively expanded into large systems. This is because such units are frequently designed to minimize costs. While, for instance, it may be possible to add another few hundred to a few thousand bytes of memory, or to add a few I/O interface circuits, it may not be feasible to expand the unit much beyond that level of capability unless one is capable of constructing electronic equipment from "scratch." The reasons that such units typically cannot be expanded substantually is because they lack suitable signal buffers and amplifiers (necessary to drive large amounts of memory, I/O interfaces, etc.). Or, they do not have sufficient memory address lines to address memory beyond a certain address value. Or, they lack I/O port selection circuits, or have had similar design cutbacks instituted in order to provide an ultra low-cost unit intended for evaluation/testing/educational purposes. The reader should not make the mistake of obtaining an evaluation or essentially tutorial

system if one has the eventual plan of building it up to a large multiple-purpose facility. There is simply a better way to proceed if such is the case, as will be discussed shortly.

The kind of tiny microcomputer system being discussed has plenty of applications and is an excellent unit for a beginner to consider when entering the field. The comments made in the previous paragraph are not meant to be negative or derogatory in regards to the tiny microcomputer. They are meant simply to keep the reader from attempting, through inexperience, to use them beyond their intended capability. The fact is, such units can keep a newcomer to computers quite occupied learning programming at the machine level (and possibly at a limited level of high level programming) for a considerable amount of time. If one keeps the types of projects one indulges in using the unit in proper scale and prospective, such systems can be constant and intrigueing sources of entertainment and perform countless useful tasks. In fact, such systems are practically ideal for people who are primarily interested in using a computer as a controller, and for people who want to emulate or simulate electronic logic networks using a microcomputer. (The capabilities of these types of units, some readers may discern, also means that quite often they can be pressed into useful service as a subsection of a larger computer system. For instance, such a unit might be dedicated to controlling a complex piece of machinery. It might be used to intercept, sort, reduce, or correlate information going to/from such a machine. Or, it might store and then transfer data it obtains in regards to its specific tasks to another more powerful microcomputer!

Small systems of the type being discussed here virtually require that the operator program them using machine language techniques. Some of the evaluation systems are so small in terms of memory capability that they cannot support machine language programming aids such as Editor and Assembler programs. This means the user must rely entirely on hand program assembly methods such as was described in Chapter Three. This is a word to the wise who do not feel they would enjoy the type of detailed work required to successfully develop and implement computer programs in this manner. Because of the fact that programming will be done extensively at this detailed level, it is most important that the unit be equipped with a satisfactory Monitor program. A Monitor program, as mentioned in

Chapter Three, allows an operator to use I/O devices (such as an input keypad and hexadecimal or octal digit displays) to perform various types of operations in a convenient manner. In a small evaluation system the Monitor program is generally provided on ROM elements so that the entire range of Monitor-provided capabilities is available as soon as power is applied. (This is definitely preferable over, for instance, merely having a bootstrap loader available in such a small system. This is because in such a system, the more provided on ROM elements, the more room one may anticipate having available in whatever read and write (RAM) memory is available in the system.) A good rule of thumb is to look (in a tiny evaluation system) for the Monitor program with the most features. Typically, Monitors with the most capabilities will provide the largest amount of ROM as part of the standard equipment package! Differences between available units vary greatly in this department. Some systems are supplied with just 256 or 512 bytes of ROM containing Monitor routines. Others come with 1 or 2 K versions of Monitors. The latter invariably has considerably greater capability. (Note: the abbreviation K, as most readers may know, is often used to signify the value 1,000. However, since computer systems are based on the binary numbering system and use numbers that can be represented as powers of two, the abbreviation K when associated with computers actually represents the value 1024. This figure is the result of the calculation 2^{10} (two raised to the tenth power). Thus, a 1 K Monitor program would actually occupy 1024 bytes of memory. A 2 K version would occupy 2048 bytes.

What kind of applications might a small evaluation-type unit serve well? Besides its use as an educational unit for which it is primarily intended, such a system (equipped with a good Monitor program on read-only-memory, with say, 1 to 2 K of RAM elements) might be applied in the following areas: playing of a wide assortment of games; control applications, such as running a model railroad, a burglar alarm system, or various types of machinery; instrument monitoring and collection/sorting of data from equipment such as counters, digital voltage or frequency measuring devices, or various kinds of scientific apparatus; code conversion - such as coupling Baudot coded devices to ASCII coded units; a network control or routing device to accept messages from one source, then transmit them to another; logic replacement - a microcomputer can be used to substitute for large amounts

of logic circuitry typically developed using complex integrated circuit networks; operation as a calculator; use as a peripheral controller in a larger computer system; and so forth. The range of uses is truly limited primarily by one's imagination and programming skills!

Small training or evaluation units of the type just discussed, along with a suitable keypad and seven-segment type of display, a ROM containing a satisfactory Monitor program, and with anywhere from a few hundred to a few thousand bytes of RAM elements installed, are available from a number of manufacturers for in the range of $100 to $300. By including another $50 for a small power supply, the reader can see that with a relatively small investment, one can get started in microprocessors. (Remember the limitations against expansion likely to be encountered.) It is a good way for many people to get "hands on" experience, particularly if they already consider themselves electronic enthusiasts and are willing to deal extensively at the machine language level.

The next step up the microcomputer ladder, economically speaking, is to a medium-sized system outfitted primarily for "personal" applications. (The word "personal" referring essentially to use by individuals on a "one-at-a-time" basis, regardless of the environment. It does not necessarily imply that the system is dedicated solely to use by just one person! Indeed, businesses, schools, etc., may well utilize such systems by allowing their use by a number of individuals.) The primary differences between this type of a system and the tiny evaluation units might be the types of I/O devices likely to be used, the amount of memory typically installed, and the expansion capabilities of the system.

A good, medium-sized personal computer system, in the opinion of many, should be modular in design. It should be capable of being readily expanded by the addition of memory and I/O interface modules. It should be capable of being expanded to the point where it contains enough memory to adequately support a high level language. ("Adequately support" meaning to be able to hold the high level language interpreter and still provide sufficient memory space for a user's high level program of a practical and useful size to reside in memory.) It should be able to accept I/O interfaces to high speed

bulk storage devices (such as higher speed magnetic tape units or floppy disk units). In this regards, it should also be able to control two or more such devices to allow the transfer of information from one such device to another.

Preferably, a good general purpose, medium-sized personal computer system should be capable of operating with peripheral devices in what is known as a DIRECT MEMORY ACCESS mode (often abbreviated as DMA). DIRECT MEMORY ACCESS operation refers to a special mode of operation. In this mode, the CPU in a computer system temporarily relinquishes control of the memory address and memory data signal lines. (Those lines select specific locations in memory and carry the information to/from addressed locations.) Once the CPU has relinquished its control over those lines, a peripheral device is allowed to exercise its control over them in order to select desired locations in memory. The peripheral device then directly transfers data to/from the selected locations. This type of operation results in the peripheral device being able to transfer information to the computer's memory banks at a much faster rate than if each byte of data had to be transferred by following a series of instructions under control of the CPU.

A medium-sized general purpose system should be capable of at least some form of INTERRUPT processing. Preferably, it should be able to respond to several levels or types of INTERRUPTS.

What is INTERRUPT processing? Interrupt processing is analogous to a postman being interrupted while delivering the mail (as described in Chapter One). Suppose, as a postman was placing mail in a row of mailboxes, someone walked up and tapped him on the shoulder (interrupted him). The mailman might complete putting mail into the box he was currently filling, make a mental note regarding where he left off work, and turn to the person who had tapped him on the shoulder to see what the interruption was all about. Suppose the person who had interrupted the mailman had a letter for the mailman to deliver further down his route. The mailman would take the letter and place it in his mailbag. At that point, he would be through processing the task for which he had been interrupted and could return to the original job he was working on which was filling a row of mailboxes. After recalling where he had left off that task,

he could continue on with his job of filling the next mailbox. One could say that the mailman had "processed an interrupt!"

For indeed, when a special line on a computer system so-equipped receives a signal referred to as an INTERRUPT signal, the computer will behave in a fashion similar to that just described for the mailman. Upon detecting the presence of an "interrupt" signal, the CPU will finish executing the instruction it is currently processing. It will then "save" various types of status information, such as the contents of the program counter (the address in memory where it was working), contents of CPU registers and CPU flags that might be altered by processing the interruption, and so forth. Typically, the CPU then performs a JUMP or CALL type of directive to a specified address. There it expects to find a series of instructions that will process the peripheral device responsible for generating the interrupt signal. After the instructions have been processed to take care of the device that initiated the signal, the computer is able to return to its original programmed operation. It does this by restoring the contents of the CPU registers and flags that had been "saved" as well as the original contents of the program counter. (Note that typically those registers whose contents are "saved" while processing an interrupt are simply placed on a "stack" in memory. The "stack" is pointed to by the CPU's stack pointer register (refer to Chapter Two if necessary for a refresher).

Why is the capability of being able to process an interrupt signal so important in a general purpose computer system? Because being able to respond to an externally generated interrupt signal allows a computer to perform calculations, or process data, while waiting for a relatively slow I/O device. It then does not have to waste time devoting its capability to constantly monitoring a slow device until it is ready for a transfer operation. Remember, a microcomputer is able to execute an instruction in but a few millionths of a second. Many I/O devices may take several thousandths of a second just to transfer an eight-bit character. A CPU could possibly execute a thousand instructions in that time period. Or, to illustrate the concept even more dramatically, consider the case of a magnetic tape unit taking 10 or 15 seconds to search a long tape for a file gap. A lot of time would be wasted if the CPU had to monitor the I/O port waiting for a "file gap detected" signal line to change state. If, however, that line

was connected as an interrupt line, the CPU could perform other operations until the interrupt line was activated, thus signaling that a file gap had been detected. In essence, interrupt capability makes a microcomputer more time efficient.

Some microcomputers have the capability to handle several levels of interrupts at one time. That is, if it is handling one interrupt, it can still safely handle a second - or even a third on top of that. Often such systems have interrupt lines that have a priority ranking so that only certain lines can over-ride other lines. Other microcomputers with interrupt capability require that any priority system be controlled by software. Unless a user plans on using a personal system for special purposes heavily involved with controlling I/O devices, a relatively simple interrupt capability will be satisfactory in most cases. Inclusion of at least some such interrupt capability can come in handy in a medium-sized system.

The reader that starts reading about various types of systems in the trade journals, procures manufacturers' literature, and so on, will invariably run into the discussions of a microcomputer's SPEED of operation. The reader is advised to try and avoid attempting to rank the various types of computers according to specifications related to operating speed. For one thing, the specifying of speed in a microcomputer is often given as it relates to the computer system's main CLOCK. A CLOCK in electronic circuitry is simply a relatively stable continuous string of pulses that is used to synchronize the activities of all the various modules or sections of a computer with one another. Frequently, a computer's clock frequency (for instance, figures such as 500 kilhertz, 1 megahertz, 4 megahertz) or the system's clock period (time between two pulses or cycle length, represented by figures such as 500 nanoseconds, 1 microsecond, etc.) will be specified. Or, a CPU will be advertised as being capable of operating at a clock frequency of "such-and-such." Do not be overly impressed by such figures. For one thing, initial clock frequencies provided in computers are often divided down for technical reasons to much lower rates. Maximum figures at which CPU units may be advertised as capable of operating at are often not used by system designers. Beyond that, quite often, while one portion of a system may be capable of operating at one rate of speed, another portion may not be capable of such operation and will thus effectively force the former section to have

to wait, thus negating its touted speed advantages.

This conflict in operating speed capability is frequently encountered in relation to memory elements in a computer system. Memory elements are often sold and graded according to their ACCESS TIME and other parameters. ACCESS TIME for a memory element indicates the amount of time that passes between placing an address on the address lines of a device and the time when data on the data lines from that device can safely be assumed to represent the contents of the location addressed. Access times are typically measured in hundreds of nanoseconds (billionths of a second) or in microseconds (millionths of a second). As a rule of thumb, the faster the access time for a memory element, the more expensive that element is to produce. Manufacturers of memory modules often buy memory integrated circuits according to their access time and other parameters that impact costs. Thus, some systems containing a relatively fast CPU (and stressing this point in advertisements) may contain memory elements that cannot match the processor's speed when it comes to memory access operations, thereby forcing the CPU to effectively wait for memory elements to be able to deliver (or accept) data! For instance, some CPU devices are able to transmit an address (from the program counter) and then continue operation to be in a position to fetch an instruction from that address in, say, 500 nanoseconds or less. If, however, the memory elements in the system are not able to decode and allow access to the addressed location in that amount of time or less, the CPU will simply have to enter a so-called "wait state" until the memory elements have had time to respond!

While talking about system speed, it might be interesting for the reader to learn about the difference between two types of RAM elements referred to as "static" versus "dynamic" devices. For technical reasons, "dynamic" elements generally cost less and have faster access times than "static" ones. However, they have a drawback which will be pointed out shortly.

Static memory elements are so-named because once something has been stored in them, it will remain there (unless power is removed). In other words, once data has been stored, the memory is "static" and unchanging unless it is deliberately altered (written over) or power is removed from the device.

Dynamic memory elements, on the other hand, are not so simple. They require constant activity (thus the nomenclature "dynamic") in order to retain information. That is, the address lines must be repeatedly activated, then deactivated, in order to "refresh" the data contained in the memory cells. Of course, this requirement for activating the address lines to the dynamic memory elements is met whenever the CPU accesses a particular location in memory. Unfortunately, in even a small system, the CPU will not normally be able to activate all the possible addresses within the required amount of time. (Usually, each cell or address in a dynamic memory element must be refreshed every one or two thousandths of a second.) This generally means that separate memory refresh circuits must be provided in the computer. Such circuits are able to address memory elements in essentially a DMA (direct memory access) mode. Using this mode, these circuits constantly exercise the address lines going to the dynamic memory elements in order to perform refreshing operations. If, for any reason, refresh operations are delayed beyond the time in which a cell can retain data (which is just a few thousandths of a second), the status of a cell may be altered. Thus, the refresh operations must go on constantly.

The requirement for constant refreshing of dynamic memory cells constitutes the drawback against their use. Their lower cost is somewhat offset by the additional circuitry that typically must be provided in a computer system to perform the refresh operations. Secondly, even though the actual access times of dynamic elements may be faster than static elements, the overall operation of a computer that utilizes them may not be faster than one using static elements. This is because the refresh operations being performed by the special circuitry in a DMA mode effectively "steals" time away from CPU operations, making the CPU wait between operational cycles when refresh operations must be performed.

A final reason why operational speed claims for a microcomputer system intended for personal general-purpose use may be taken with a "grain of salt" is because, for all practical purposes, virtually any microcomputer on the market today is "fast enough." Virtually all these machines have CPU devices that operate in the range where the time it takes to perform an instruction is measured in millionths of a second. Whether one model takes a few millionths of a second longer

than another is not going to make much practical difference to most users. Where speed differences may begin to mount is when one plans to make extensive use of high level interpreters or run very large data manipulating programs (such as sorting programs) on vast amounts of data. For the average system user, the difference of a few seconds in the performance of a sophisticated high level calculation is not a major concern.

If one must be concerned with speed of operation in a computer system, one should not look for the system clock specification! Instead, one should look for specifications telling how long it takes to perform specific types of instructions in a particular system. This is the key figure of performance, and the one that can take into account all the factors that may affect time consumption. (Note that one should look for system figures, not just CPU figures. Remember that while a CPU device might be capable of performing an operation in a given amount of time, when installed in a system that, say, utilizes dynamic memories that must be refreshed, the length of execution time may be considerably increased.) As a rough guideline, the reader will find that typical "high performance" microcomputer systems execute various types of instructions in from 1 to 10 microseconds. Lower performance units (often based on CPU devices that are more suitable as controllers rather than for use in general-purpose systems) typically perform instructions in 10 to 50 microseconds. It might be worth paying a little more to get a system that performs in the first category, but to pay premium prices for differences in that class is questionable for most users.

A CPU parameter that the purchaser of a medium-sized system might want to check on is whether the CPU uses a stack pointer register that points to addresses in memory. Or, if the CPU has an "internal" stack. In a unit having an "external" (in memory) stack, the stack pointer should be capable of being manipulated (for instance, incremented and decremented, set to a particular value, added to values in other CPU registers) in various ways. It should especially have the capability of being able to be set to a desired location in memory. This feature will allow multiple "stacks" to be established in memory which can be very useful in various kinds of programs.

Some types of microprocessor CPU devices, instead of providing a

stack pointer register that addresses locations in memory, provide a limited stack (group of registers) on the CPU chip itself. A major drawback to this technique is that such a technique restricts the size of the stack to the number of registers provided for that purpose on the CPU chip. While such an arrangement does allow standard subroutining operations to be performed (which require that the contents of the program counter be stored on a stack), it does not provide the flexibility that may be obtained when stacks may be formed in memory. They also do not allow the depth of nested subroutining (where one subroutine in turn calls another, which calls another. . .etc.) that may be desirable in certain situations. When a stack pointer register is available, thus allowing memory to be used to form stacks, the depth (length) of those stacks is essentially limited by the amount of memory provided in a system, allowing orders of magnitude greater depth!

It is possible for a person to work with a general purpose medium-sized system using front panel controls. Or, to use simple keypads and seven-segment displays as is typically done with evaluation type units. However, most people investing in a medium-sized system will want to have the use of somewhat more efficient I/O devices. A medium-sized system invites the use of full alphanumeric keyboards for normal operator input and a printing or cathode-ray-tube display for outputting data. A medium-sized system can hardly be called complete without some form of bulk storage I/O, such as a magnetic tape or floppy disk unit.

A "general purpose" system today can hardly qualify for that designation with a complement of only 4 K RAM elements. Such a small amount of memory would severely restrict the scope of the system. While one might be content to initially start with 4 K of RAM elements during the familiarization stage, a more practical system would have at least 8 K of memory, and 16 K would be a better figure to work towards. Beyond 16 K of RAM elements, one can begin to work around a good high speed bulk storage device (such as a floppy disk) when dealing with programs that must manipulate a lot of data. However, the capability to expand upwards to 32 or 64 K of memory is a good option to have available in any system that one intends to put to good use over the long term.

A microcomputer system with, say, 8 K of memory can usually adequately support Editor and Assembler programs that will allow a person to efficiently work at the machine language programming level if desired. The use of such aids multiply by a significant amount the efficiency of working at that level over hand assembly methods. (Provided that the system has the bulk storage capabilities required to rapidly load in the Editor and Assembler programs, as well as process the source listings and object codes produced!) The prospective purchaser of a system interested in that type of programming should ascertain that Editor and Assembler programs for the CPU being utilized are available. One should also determine the amount of memory required to hold and comfortably utilize the programs.

The person getting a medium-sized system will probably want to make sure that the CPU type chosen is one for which versions of one or more higher level languages such as BASIC have been developed and are readily available. Most BASIC interpreters designed for microprocessors will reside in 8 K of memory while leaving some room for a user's high level language program in memory. (There are also miniature "limited" versions of BASIC commonly available for some types of CPUs that will execute in 4 K of memory.) Some software firms also produce versions of BASIC that have special capabilities that require more memory for the interpreter (such as 12 K) which are often referred to as "Extended BASIC" interpreters. Remember, however, that a so-called "8 K version" of a typical interpreter will only leave 1 or 2 K bytes of memory in an 8 K system for storage of a user's program. Many users prefer to have 12 or 16 K of memory when running such a version in order to provide a decent amount of memory for a user's high level program.

How does one really know how much memory to provide in a computer system that will be used primarily for executing high level language programs? It is not difficult to get a rough idea of one's requirements. There are many books available today describing various kinds of programs written in, for instance, BASIC programming language. A person can browse through such a book selecting the types of programs one might find of particular interest. Or, alternately, one can study the information provided in Chapter Four of this book, or perhaps examine a book that describes BASIC language in more detail. Then one can work up representative programs that

one might want to use. In any event, once one has obtained a source listing of the high level program one wants to run, one is in a position to estimate the system memory requirements needed to run the program.

First, one needs to count up all the characters in all the statement lines contained in the program. (One can soon learn to do this in a rough fashion by estimating the number of characters per line according to line length.) Next, one must allocate memory space for all variable values that will be stored by the program (meaning, in this instance, numerical values associated with variable names). A good rule of thumb here is to allow about eight bytes per variable element assigned in a program. Finally, one may need to allocate storage for alphabetical strings that one might input to a program (such as the number of characters in a name and address).

As an example, suppose a reader is interested in running a program that consists of 40 statement lines, uses 8 variable values, and does not use any alphabetical strings. The reader finds that there is an average of about 40 characters on each statement line. Statement lines constitute the source listing of the high level program. They must reside in the computer's memory so that the BASIC interpreter may scan and decode their meaning. The reader probably realizes that the statement lines would typically be stored in the form of ASCII coded patterns. Hence, each character in a statement line will require exactly one byte of memory in the computer. Multiply 40 lines by 40 characters per line and one knows that 1600 bytes of memory will be required to store the high level program itself! Next, multiply the number of variable values in the program by the suggested value of 8 bytes per variable. (The 8 bytes per variable is derived from knowledge of how variable names and their associated numerical values are typically stored in "tables" in a computer's memory when using an interpreter.) For the example: 8 times 8 yields 64 bytes required for storage of variable values. Since the example did not use alphabetical strings, the total memory requirement for the storage and execution of that high level program would be 1664 bytes.

Suppose a person wanted to run the example 40 statement line program in a system having 8 K of memory. Suppose the BASIC interpreter being used resided in 6,500 bytes of memory. Can the

person load and execute the program? Remembering the fact that in computer circles, 8 K really means 8,192 (8 x 1024), one can quickly determine that:

$$8192 - 6500 = 1692$$

Yes! With just a few bytes to spare!

These types of easy to do calculations can also provide some valuable insight for the reader regarding memory requirements. Take a look at what happens, for instance, if the user wants to load and execute a BASIC program that has 300 statement lines - again averaging 40 characters per line - and that uses, say, 150 variables:

$$(300 \times 40 = 12{,}000) + (150 \times 8 = 1200) \text{ or a total of } 13{,}200$$

Now the reader may observe that the program itself needs about 13 K of memory. Add that to the 6.5 K for the interpreter, and one may conclude that one will need a system with 20 K of memory!

Or, look at the impact of adding in the use of alphabetical character strings. Such would be the case, for instance, if one wanted to process a list of names and addresses using a high level language. Consider a hypothetical program containing 100 lines of 40 characters average length, say just 20 variable values, but expecting to input and process perhaps 200 names and addresses - each name and address averaging 80 characters. Look at what the memory requirements would be if the program was executed in a straightforward fashion:

$$(100 \times 40) + (20 \times 8) + (200 \times 80) = 20{,}160$$

Including the 6.5 K for the interpreter program, one is in need of a system having near 28 K bytes (for the sake of rounding to the nearest convenient 4 K module of memory elements). One point being made with these calculations is that manipulating data in alphabetical form, a common requirement in many applications, takes a lot of memory.

Another point to be made is that at some juncture, one must consider breaking large programs up into subsections and using bulk

storage devices to load in appropriate sections of a program or data, etc. This technique, at the sacrifice of execution time since such transfers cause delays, allows one to deal with programs that might far exceed the actual active memory capacity of a computer.

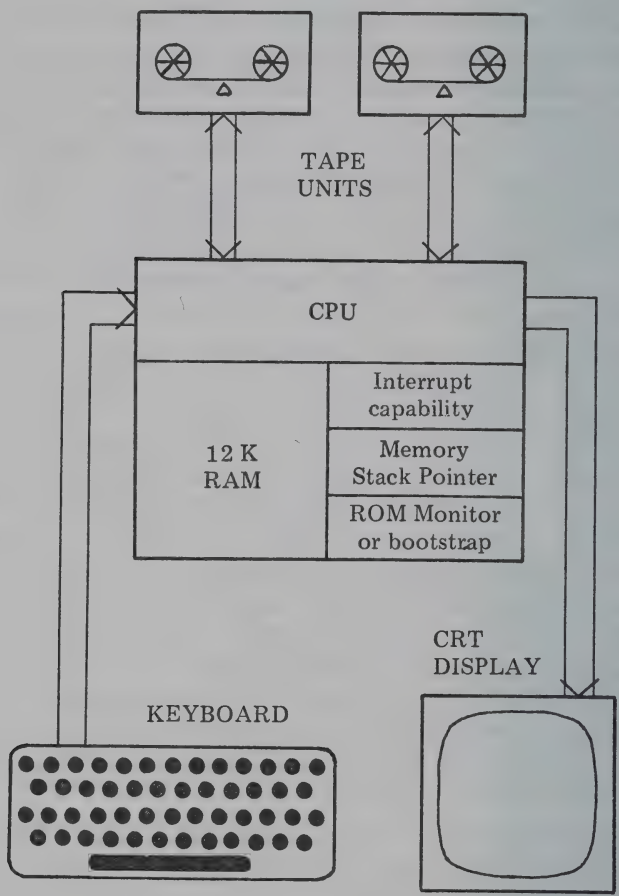
In a small evaluation-type system, it is important to have a good Monitor program on read-only-memory elements. In a medium-sized system it is important to have a good Monitor program. However, a medium-sized general purpose system will typically be equipped with a relatively sophisticated I/O arrangement, such as a keyboard and CRT or printing display. It is also likely to have relatively sophisticated bulk storage facilities. Thus, it is generally feasible to provide a suitable Monitor program that resides in RAM elements. Because of this, in a general purpose system, one may consider getting by with just a small bootstrap loader routine on ROM elements. Whenever power is applied to the system, the bootstrap loader may be instantly utilized to load in a Monitor or high level interpreter from a bulk storage device into read and write memory elements.

Of course, if one has the financial resources, one may equip a general purpose system with a variety of programs on read-only-memory elements. It is possible to obtain complete programs such as Editors, Assemblers, and high level interpreters on ROM, PROM, or EPROM devices, or to obtain listings of programs that can be installed on read-only-memory devices. (Large programs such as the types mentioned must generally be specifically designed to be able to reside in ROM elements. Care in the creation of such programs must be taken to ensure that all operations involving the moving of data, etc., take place in memory address locations that will contain RAM elements. Such programs invariably must have access to some RAM locations for storing temporary data.) A person planning on purchasing programs for specific purposes that may want to eventually install the programs on PROM devices should ascertain whether or not the program was designed to reside in such elements. If it was, one may reasonably expect to pay a higher price for it than for one that does not have that capability. It takes more effort in many instances to make a program "ROMable." Remember, if a program is designed to alter memory locations in any way, the locations that are to be altered cannot be at addresses assigned to read-only-memory devices!

What will a medium-sized general purpose microcomputer cost? It depends, of course, on whether the system is obtained in kit or assembled form, the amount of memory, the amount of support provided by the manufacturer, and so forth. Consumer quality systems for essentially non-commercial environments can start in the $300 to $400 range for kits with a limited amount of RAM. This might include, in a typical modular system, a CPU, 4 K of RAM, one or two I/O interfaces, a backplane buss unit, a power supply, and perhaps a limited version of a bootstrap loader or Monitor program on ROM. A MAINFRAME (computer without external equipment) in this lower price range would probably not have a sophisticated front panel or manual control module. Adding a few hundred dollars can get a system with front panel controls and probably more expansion capability. Commercial grade mainframe units can reach the vicinity of $1000 or more for similar capability. Increasing the amount of RAM usually costs in the vicinity of $100 to $200 per 4 K module, depending on the system, speed of the memory elements, etc. Referring to the previous chapter, the reader must remember that the cost of I/O devices such as a keyboard, display unit, and an inexpensive bulk storage device such as an audio tape unit and control interface, must be added to the mainframe costs. Additionally, one must sometimes pay additional for various types of programs that might be desired. One must also pay for any "frills" one might want, such as ROM elements, which may not be included in the regular price of a unit. All in all, a practical general purpose medium-sized microcomputer system with say, 8 to 12 K of RAM, along with the appropriate I/O devices to make operation convenient (a keyboard, CRT, and audio tape cassette), can run in the vicinity of $1000 to $2000. The price, as indicated, can vary substantially, depending on a person's preferences. An assembled and tested system with "frills" will typically be in the high end of the range, a "bare-bones" kit in the low part of the range. (These figures are simply provided as "ball park" figures for the reader to take as beginning reference points.)

What can one practically expect to be able to do with a medium-sized microcomputer system with 8 to 12 K of RAM and suitable I/O devices as indicated? Quite a bit. In addition to all the types of activities one might do with the tiny evaluation units as discussed earlier, a general purpose system would typically be able to do the following tasks: sort and store data for coded retrieval; play rela-

tively sophisticated games for entertainment and education; serve as a word processing or text editing tool, such as a "memory typewriter;" maintain and calculate various types of financial records of small to medium size - say for personal use - such as balancing checkbook banking statements, preparing tax statements, and maintaining expense accounts; operate external equipment such as instruments or machines while accepting and processing data from such



BLOCK DIAGRAM OF A GENERAL PURPOSE MEDIUM-SIZED MICROCOMPUTER SYSTEM SUITABLE FOR PERSONAL APPLICATIONS

equipment; and a myriad of other types of activities limited primarily by the imagination and programming time and skills available to the prospective user. From studying the previous chapters, the reader should have a good feel for the kind of potential a system with 8,000 to 12,000 bytes of memory has, whether the system is programmed using a high level interpreter, or using machine language methods. A machine with that amount of memory can adequately support the programming tools that enable the machine to help the user program it. It can support the peripheral devices that will allow the user to communicate conveniently with the computer. Those factors provide the big difference in utilization between such a system and the evaluation-type units discussed previously.

The final type of system to be discussed is the large microcomputer system capable of performing small business applications. Actually, such a system may be considered as essentially just an expansion of a medium-sized system. However, there are several factors which are worth discussing for the person considering acquiring a microcomputer-based unit for such applications.

It seems that a substantual number of people interested in having access to the use of a "personal" computer system want to, for one reason or another, justify the acquiring of the system for "business reasons." Well, there is nothing wrong with acquiring a "personal" computer system for business applications. The problem that people frequently seem to get into, however, is that they attempt to use such systems for business purposes without applying business reasoning and safeguards! Believe it, more than one company has gone bankrupt because they got carried away with applying computers without applying business techniques and safeguards. (In fact, entire books have been written on the subject. No such in-depth attempt to cover the subject will be made here. Rather, several salient points will be made to point out that this is an area where a person should venture with caution.)

First, while the field will definitely be covered increasingly in the not too distant future, there is currently relatively little in the way of systems and software specifically developed for, and economically attractive to, the small business operation (businesses having from a few to perhaps 20 employees) based on the use of microcomputers.

The field is really in its beginning stages, and in many respects actually lags the development of "personal" computer systems intended primarily for non-commercial applications. What is available, in many cases, must be considered as relatively new and untested by time. Thus, persons planning on becoming involved in the area should consider themselves "pioneers" and take appropriate precautions! Chances are one is going to have to become intimately involved in all areas of such a system's operation; from management, to programming, maintenance, and operation!

The dividing line between a medium-sized "personal use" system and a large "business-oriented" system is a rather arbitrary one. However, a business oriented one should definitely have good marks in the areas of quality (reliability) and high speed I/O capability. A third distinguishing difference might be the amount of memory available in a system. A business oriented system is likely to need a goodly amount of both mainframe and off-line bulk storage memory.

Reliability in a personal computer system is a parameter that often can be sacrificed somewhat in favor of practical considerations, such as lower equipment costs. Should a tape unit malfunction while reading in a program, it is generally no serious matter for a hobbyist user to take a few additional minutes to reset controls and start the tape read process over again. The enthusiasts can also typically spare the time to, for instance, make duplicate copies of magnetic tapes containing important programs. If the system goes on the "blink" for a few days, the casual user can probably get along without losing any money. In fact, such a user may even enjoy an equipment malfunction for the sake of being able to troubleshoot and repair the equipment.

A malfunction in a computer system dedicated to business applications can spell disaster. The loss of a magnetic tape containing files of a business' most valued customers is no small matter. A computer out of operation for a few days or a week while waiting for a serviceman to come and repair it is no laughing matter. This is especially so if it is in constant use for such purposes as: verifying customer orders; keeping track of inventory; performing invoicing and billing operations; or calculating paychecks!

Increased equipment reliability frequently means increased costs.

Equipment intended for commercial applications and meeting commercial operating specifications commands higher prices. The purchaser of a system to be used primarily for business applications is well advised to verify the availability of service or maintenance policies and agreements, or be prepared to assume such responsibilities. One might also want to look into obtaining special features for a business oriented system such as having battery-powered backup for the read and write memory elements in the event of a power failure. In short, one needs to pay close attention to quality and performance details affecting reliability that one might tend to skip over or ignore in a system for non-commercial purposes.

The performance of a bulk storage device in a personal system in terms of speed and amount of storage is also a parameter that can often be sacrificed to achieve lower cost. One might be perfectly happy to use an audio magnetic tape system to hold programs and data in such a system. One might also be content to get along with just a single bulk storage device in a non-commercial environment.

Such is generally not the case in a system dedicated to business applications. An audio magnetic tape system, except for such purposes as initially loading programs, particularly as a backup unit, is simply too slow to earn its keep in most commercial applications. A business system generally needs the capabilities of a high quality digital tape unit which can achieve superior speeds and increased storage capability over that possible with audio tape units. A commercial system also needs, in most instances, two or more bulk storage devices on-line at one time. This is because many business programs involve operations (such as sorting and merging files of information) that cannot be accomplished in a practical manner using just one bulk storage device. This is because when doing such operations, it is frequently necessary to process information as it is received from one (or more) input device(s) and output it to another device.

Most people planning on building up a small computer system for business applications would be well advised to settle for nothing less that a system with two or more floppy disk units instead of magnetic tape units. The greatly enhanced speed of operation of the floppy disk units over most tape drives is usually well worth any difference in price. (Remember, the price differential between good commercial

quality magnetic tape units and floppy disk systems is likely to be relatively small. This is not the case when comparing between audio tape systems and floppy disk systems. There, the price differential can be substantial.) The only sacrifice that might have to be made for such a choice would likely be in terms of the total number of bytes of information that could be accessed under computer control at any one time. Dual floppy disk drives (standard size) would typically be able to store 500,000 to 1,000,000 bytes. Some digital tape drives might be able to pack twice that amount or more on tape cartridges. However, since the storage medium (diskettes) can always be inserted as needed, the limitation is certainly not a severe one in most instances. Assuming, for instance, the use of 80 characters (bytes) per entry, a dual floppy disk system with 1,000,000 byte capacity could store some 12,500 entries and access any one in about one second if the information was properly organized on the diskettes.

From the discussion presented earlier illustrating the benefits of having large amounts of RAM elements available in a system, the reader may surmise that in business applications involving the manipulation of large amounts of data, one may always use increased amounts of RAM elements. The more read and write memory in a system, the less often the system is likely to have to call on time-consuming I/O operations in order to bring in new blocks of data.

The small business system is also an instance where one might want to consider paying some attention to, and possibly paying a premium for, increased speed of CPU and system operation. Generally, in a business situation, time is money. The seconds that a higher speed system might cut off performing various kinds of calculations over the years might be worth paying for at the time of initial purchase.

The prospective purchaser of a microcomputer system is well advised not to mix enthusiasm for the field with business decisions in planning such a system! A person all excited by the prospects of "personalized" computer systems can sometimes mentally justify the purchase and applications of such units in situations where they do not belong on business grounds! Computers make business sense when they are used to solve complex, repetitive problems, or where they manipulate relatively large amounts of data in ways not easily

accomplished by people. They are sometimes applied in situations where they do not belong - usually being placed in that position by over-enthused (and usually well meaning) computer worshippers.

For instance, more than one prospective computer purchaser has proposed the use of a computer for the following type of task: The computer is to maintain a "customer file" of clients and an indication of the client's credit rating and last date of ordering. Whenever an order arrives, a person enters the client's name on a terminal. If the customer has a previous account, the computer will display the account's status and allow the operator to update it. If the account is new, it is added to the data base. Good use of a computer system, right? Wrong. One can accomplish the same objectives with a conventional card-file system. In the amount of time it takes to type in a client's name on a computer terminal, a person can locate and extract a file card in an alphabetized system. For a data base of a couple of thousand customers, say, the file card system would be far more economical than the computerized system, both in initial and maintenance costs! Remember, it takes just as long to type information using a keyboard terminal connected to a computer as it does to type the information on a file card!

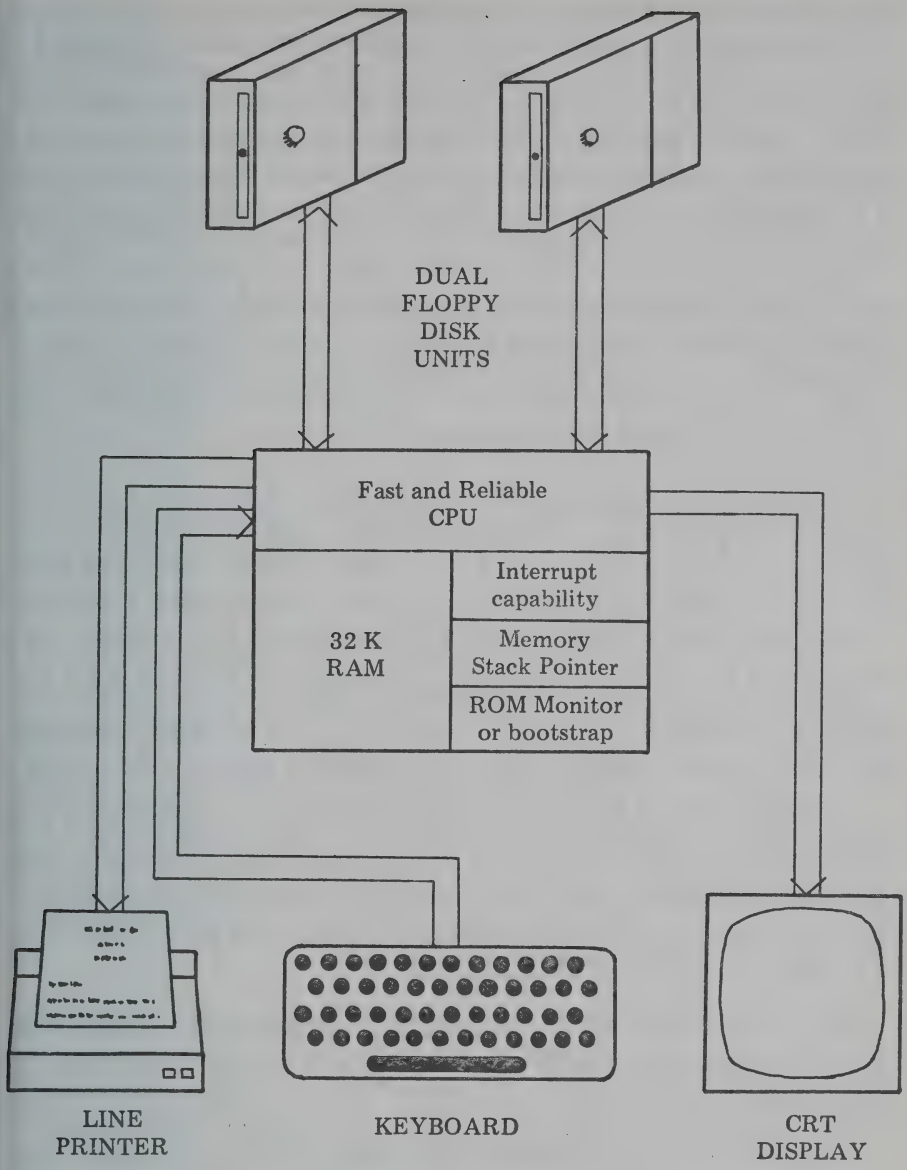
Where would a computer system start to make economic (business) sense? It would make sense in the situation just presented if the computer was used to do more work! Suppose, for instance, that at the end of each month it was necessary to go through the file of customers and send "past due" notices to all whose accounts were in arrears. It would take a person quite some time to go through all of the accounts if they were on file cards, checking to see if their accounts were past due, and then typing out a past due notice. A small computer system, on the other hand, with each entry stored on a fast-access peripheral (such as a floppy disk) could whiz through the checking process in a few minutes. Any accounts found to be past due could result in the computer printing a billing statement right on the spot - ready for inserting into a envelope (say, with the name and address showing through a glassine window) and mailing! Not only that, but the system could be directed to build up a "collections report" for management as it did the billing task. It could then display summarized information for review - such as the number of accounts past due, the amounts that were 30 and 60 days

past due, and so forth.

The same data base might be used to generate mailing lists by categories. Whenever it was desired to notify customers interested in certain types of products (whose interests had been appropriately recorded on their files), the computer could be directed to search the data base and print address labels. By storing other types of information on each customer's record, the computer could be used to sort out and correlate other types of information that might be extremely useful in the operation of a business. Trends towards certain types of products, periodical buying patterns, or correlation of certain products with, for instance, consumer age groups might be ascertained using the capability of a small system. In short, as a general rule, the more data and information that needs to be stored and processed, the more valuable a computer becomes.

The business user must make plain, no-nonsense business decisions to determine if a computer system can be put to effective use, or whether it might simply be an expensive luxury item. It seems that one of the easiest things to forget when planning a business system is that "the data base does not establish itself." Someone has to physically input all the data into a computer system - typically by sitting at a keyboard and typing in the required information. If the information will be used repetitively and/or in a variety of ways, or becomes the basis for numerous and/or complex operations, then a computer system will probably be worthwhile. If the information is recorded essentially just for record keeping and occasional reference, there is no need to attempt to justify a computer system on business grounds.

The potential user of a microcomputer system for business applications must always pay close attention to administrative procedures, particularly in regards to protection and backup of computerized data. Even the best of available equipment is subject to occasional failure, and any system is vulnerable to a sudden power interruption. What happens if a diskette or magnetic tape cartridge is accidently destroyed, causing several thousand records to be lost? These kinds of factors require thought and consideration. Will spare copies of critical tapes and diskettes be maintained with viligance? Will sources of data be properly maintained? Are adequate conventional docu-



BLOCK DIAGRAM OF A MICROCOMPUTER SYSTEM
SUITABLE FOR SMALL BUSINESS APPLICATIONS

ments maintained to support computerized processes? A businessman may be perfectly satisfied that a computer system has produced a correct, up-to-date financial report; however, the IRS (Internal Revenue Service) may be completely unimpressed, and they (and many courts of law) require that various computerized documents be supported by records that can be audited by conventional methods. This aspect leads to another rule of thumb: The more that a particular type of information is intended for internal use, the more it may be "safely" computerized. The further away one goes from strictly internal use, the more important it becomes to back up computerized operations using conventional methods. One can do market research and customer analysis using a computer with relative abandon. However, one should keep a close eye on operations if a computer is going to make out reports for tax purposes!

What does a business quality microprocessor system cost? A system capable of performing business oriented tasks consisting of, for instance, a relatively high speed mainframe with 32 K bytes of RAM memory, a dual floppy disk system, a video display with input keyboard, and a high speed printer for generating reports, forms, etc., can run in the $5,000 to $10,000 range (depending on quality and capability - primarily of the peripherals) for low cost systems with little or no software or applications support. System costs can double for higher quality systems with software and applications support. An important point is to shop around and ask a lot of questions! The reader seriously interested in such applications should read a few books beyond this one. Such a reader should also remember what was said at the start of this discussion: The age of the application of microprocessors to small business problems is just beginning. Those who become involved will be pioneers.

The years ahead will be exciting, fun, challenging, and probably full of surprises for those involved in applying microcomputers at the personalized level. After some thirty years of being in the hands almost exclusively of large businesses and academic institutions who were the only ones that could afford the machines, the price of true computer capability has fallen to the point where it has the potential of being available to almost anyone with a serious interest. Coming with the economic availability is a sweeping wave of realization that computers are not the horribly complex and unmanageable machines

that some misinformed soothsayers have espoused. Indeed, as has hopefully been demonstrated by the contents of this publication, computers are truly simple extensions of humankind's somewhat limited logical abilities. Their operation and control is strictly dictated by their creators. Their application, for good or bad, can only be accounted for by those who provide their programs. In the opinion of many, the best safeguard against their misapplication lies in the general dissemination of knowledge of their operation and capabilities.

If this book has whetted your appetite for the subject, why not join the tens of thousands of individuals who already have their own personalized computer systems, or the hundreds of thousands who are interested in becoming small systems owners. They are communicating what they are learning about microcomputers and how to apply them through a number of periodical publications. If the reader is interested in joining the fun and learning more about the subject, the following publications, which are devoted exclusively to the field, may be of interest:

BYTE Magazine
70 Main Street
Peterborough, NH 03458

CREATIVE COMPUTING
P. O. Box 789-M
Morristown, NJ 07960

INTERFACE AGE
13913 Artesia Boulevard
Cerritos, CA 90701

PERSONAL COMPUTING
167 Corey Road
Brookline, MA 02146

KILOBAUD Magazine
Peterborough, NH 03458

APPENDIX A

COMBINED GLOSSARY AND INDEX

The following is a combined glossary and index to facilitate the use of this book. The listing is arranged in alphabetical order. It contains words and phrases associated with microcomputer technology that are discussed in this publication. Following each word or phrase is the chapter and page number where the subject matter is introduced or discussed in detail. In some instances, several locations are given indicating that significant information is presented on the matter at several points in the book. Finally, a brief explanation of the meaning(s) of the word or phrase is given as it applies in the context of this publication. Generally, the descriptions are brief, and are meant to help refresh the reader's memory or to introduce the reader to a concept that is discussed in detail at the indicated location in the publication.

A/D - analog-to-digital; 5-48: *The conversion of an analog signal to a digital (binary) representation which may be stored in a computer. For instance, an analog voltage level of 25 would be converted to the binary value 11001.*

access time; 5-30, 6-40: *The amount of time that elapses between signalling the address of a device or element and the time when information at that location is available on an accurate or reliable basis.*

accumulator; 1-9, 1-25: *A temporary manipulating register where data may be held, operated upon so as to perform various mathematical operations, and otherwise manipulated. A register that effectively "accumulates" the results of a series of operations.*

addition with carry instruction; 2-33: *An addition operation that takes into account the condition of the "carry flag" from any previous operation at the start of the current addition operation. Valuable when performing multiple-byte arithmetic.*

addition without carry instruction; 2-33: *An addition operation that ignores a previous logic one condition of the "carry flag" at the start of the current addition operation.*

address; 1-18: *The specification of a unique location, particularly that of a "word" in a system's memory banks (typically identified by a unique binary pattern).*

algorithm; 3-3: *A series of rules or directives that when followed will result in a predictable solution, operation, or set of circumstances being obtained.*

analog signals; 5-48: *Electronic signals that closely resemble in shape or form (such as when voltage is plotted against time) an actual physical process or condition. For instance, a circuit that results in a voltage level increasing one volt each time a temperature sensor increases one degree represents an analog signal.*

arithmetic instructions; 2-32: *A general class of microprocessor instructions that result in mathematical operations (such as addition, subtraction, Boolean logic manipulations) being performed.*

ASCII - American Standard Code for Information Exchange; 3-26: *A widely accepted code used to represent alphanumeric characters consisting of 128 binary patterns that may be represented using seven binary digits. The code is commonly used to communicate with I/O devices, such as keyboards, display devices, etc.*

Assembler program; 3-55: *A program that is used to convert mnemonic representations of machine instructions and addresses into actual numerical instruction codes and addresses.*

assembling a program; 3-47: *The process of preparing a program for actual execution by a computer. Generally refers to the process of converting mnemonic representations used by programmers to actual binary patterns used by a CPU.*

audio magnetic tape storage unit; 5-31, 5-32: *Refers to a unit capable of storing computer programs and/or data on ordinary audio cassette tape using audio tones to represent binary data.*

auto-indexed addressing; 2-22: *An addressing mode which results in the contents of an index register being automatically altered by some specified amount each time such an instruction is actually executed.*

average random access time; 5-38: *The average time required to reach a specific point (or piece of data) at an address chosen at random when the unit is currently at an address which has also been selected at random. For I/O devices, the average random access time is generally equal to the worst case access time divided by two.*

backplane; 6-5: *A structure that routes wires or electronic pathways so as to form a common shared network that may be conveniently accessed by other parts of a system. The structure typically provides a passive mechanical and electronic "backbone" for a system.*

BASIC - Beginner's All purpose Symbolic Instruction Code; 4-12: *A popular high level language frequently used on personal microcomputer systems.*

Baudot code; 5-21: *A standardized code used to represent alphanumeric symbols that only utilizes five bit positions. The code is frequently encountered in older equipment. While five bit positions only allow representing 32 patterns, special "shift" and "unshift" commands allow machinery using the code to represent about sixty symbols.*

BCD - binary coded decimal; 2-36: *Refers to representing decimal digits using binary notation. A decimal digit (0 through 9) is represented using four binary digits having ascending order patterns from 0000 through 1001. The patterns 1010 through 1111 are not used.*

binary; 1-11: *Having just two possible states or conditions, such as "on" or "off" - or '1' and '0.'*

binary to hexadecimal conversion; 3-13: *Translation of a number written in binary format to hexadecimal format. A simple conversion technique consisting of merely forming binary digits into groups of four and assigning each group its hexadecimal equivalent value. The pattern for each value must be memorized when utilizing this method.*

binary to octal conversion; 3-12: *Translation of a number written in binary format to octal format. A particularly easy technique to use calls for simply forming binary digits into groups of three and assigning each group its octal equivalent value (assuming the pattern for each value has been memorized).*

bit; 1-15: *The smallest possible unit of information. For instance, a binary digit can represent two bits of information (0 and 1) or logic states. The term is also frequently used to denote a binary digit position through popular usage - hence references to, for instance, 8-bit registers, meaning a register having eight digit positions. In actuality, a register capable of holding eight bits of information need consist of only three binary cells, for eight different patterns (states) may be represented therein.*

bit-error-rate; 5-36: *A measure of the number of errors likely when transmitting information. The figure is obtained by dividing the total number of bits of information found to be in error by the total number of bits of information transmitted in the test period.*

blanked; 5-14: *Refers to the Z-axis intensity modulation of the display beam in an oscilloscope. When the beam is "blanked," the screen will not be illuminated by the electron beam.*

Boolean logic; 2-40: *Refers to a class of mathematical logic operations and related rules and formulas that are highly applied in computer operations. The class includes logical AND, OR, NOT, INVERT and related mathematical operations.*

bootstrap loader; 6-14: *A "minimum" program that will enable a computer to load itself with a more powerful or efficient program. Classically, it loads a more powerful version of itself.*

BPI - bits-per-inch; 5-36: *Refers to the number of positions per inch (typically on a recording medium) which can hold a bit of information. It is effectively a measurement of the digital recording density that can be obtained on a recording device or medium.*

branch instruction; 2-46: *An instruction that can direct the computer to proceed to a new location (alter its current program execution sequence) in memory rather than simply executing the next directive stored in the next consecutive memory location.*

buffers; 6-24: *Electronic isolators or protectors. Typically implies amplifying capability in order to enable such circuits to drive large networks of electronic pathways such as would be used when signals must be widely distributed in a system.*

bulk storage equipment; 5-24, 5-25: *Equipment capable of storing (and playing back) relatively large amounts of information from the memory*

banks of a computer. Includes equipment such as paper tape punches, magnetic tape recorders, disk units, etc.

bulk storage medium; 5-24: *The physical medium on which data is stored by a bulk storage device, such as paper tape, magnetic tape, a magnetic disk, etc.*

buss drivers; 6-24: *Electronic buffers or amplifiers specifically designed with adequate power to properly activate or control electronic circuits connected to a large number of devices.*

buss loading; 6-23: *The amount of current drain, resistance, capacitance, or other electronic parameters that may affect the characteristics or driving requirements of an electronic pathway in a circuit that supplies a number of devices.*

busses; 6-5: *Wires or electronic pathways that connect a large number of devices (modules) together in a system, generally through some type of convenient connector system.*

byte; 2-17: *A group of eight binary digits.*

call instruction; 2-51: *A type of instruction that diverts program execution to a new area in memory (sequence of directives) and yet allows eventual return to the original sequence of directives.*

carry flag; 1-35: *An indicator that signals when a register overflow or underflow condition occurs during mathematical operations with an accumulator. It is sometimes also considered as a one cell extension of an accumulator during selected operations such as the execution of "rotate" instructions.*

chips; 6-1: *A small square or rectangle of semiconductor material containing microminiaturized electronic circuits.*

clock; 6-38: *A continuous series of electronic pulses used to synchronize activities throughout an electronic system. Sometimes refers to the electronic circuitry and components that produce such a signal.*

compare instruction; 2-39: *An instruction that compares (subtracts) the value in one register against the value in another, and sets the status of various flags according to the results of the comparison operation.*

Compiler; 4-6, 4-7: *A high level operating program that converts high level source listings into machine language code.*

complement a binary number; 2-37, 2-42: *The process of changing all digits that are in the logic one state to the logic zero state, and reversing all digits in the logic zero condition to the logic one state.*

conditional instructions; 2-50: *A class of instructions that can take more than one course of action (go to different memory locations) based on the condition of a status flag at the time a particular type of conditional instruction is executed.*

core memory; 6-11: *Memory elements comprised of tiny magnetic rings or "cores."*

CPU - Central Processor Unit; 1-9: *The essential central section of a computer that controls the overall operation of a*

system. It operates by obtaining and decoding directives stored in the memory elements of a computer system.

CPU-on-a-chip; 6-1: *A central processing unit contained entirely on a single semiconductor "chip."*

CRT - cathode-ray-tube; 5-11: *An electronic tube having a screen coated with phosphor that may be illuminated by an internal electron beam. The screen of an ordinary television set is an example of a cathode-ray-tube.*

D/A - digital-to-analog; 5-48: *The conversion of digital data to an analog signal, such as the translation of the binary pattern 11001 to a signal at a level of 25 volts.*

data registers; 1-30: *Special registers provided in many microprocessor CPUs for the temporary storage of data. These CPU registers may also often be used when performing simple data manipulating operations, such as incrementing or decrementing the value of a number.*

debugging a program; 3-56: *The process of getting a computer program to operate as anticipated. Typically involves performing tests to verify proper operation, checking validity of algorithms, eliminating clerical errors made during the program assembly process, and so forth.*

decimal adjust instruction; 2-35: *An instruction that adjusts the contents of a binary formatted register into groups of BCD digits. This is accomplished, essentially, by limiting each group of four binary digits to the decimal equivalent range of 0 - 9, and providing a carry into the next four digit group for decimal equivalent values outside that range (10 - 15) that can be represented in a group of four binary digits, but which have no equivalent single decimal digit value.*

decimal carry flag; 1-36: *A special indicator available on some CPUs that signals when an overflow condition occurs in a four digit BCD group during addition operations. That is, when the binary patterns represent the decimal values 10 - 15.*

decimal to binary conversion; 3-18: *Translation of a number written in decimal format to binary format using a suitable formula or algorithm. The conversion process is often more easily accomplished by programmers using the method of translating from decimal to octal or hexadecimal and then expanding those digits to groups of three or four binary digits.*

decimal to hexadecimal conversion; 3-17: *Translation of a number written in decimal format to hexadecimal format using a suitable formula or algorithm. One conversion method calls for repeatedly dividing the decimal number (and any remainders) by the conversion base (16 for hexadecimal numbers).*

decimal to octal conversion; 3-15: *Translation of a number written using decimal notation to the base 8 numbering system.*

decrement instruction; 2-58: *An instruction that causes the value of a register to be decremented (decreased) in value, generally by a count of one. However, some machines provide variations that allow the decrement value to be specified.*

digital magnetic tape storage unit; 5-31, 5-36: *A tape recording system that utilizes the technology of storing data as tiny magnetized segments on magnetic tape. When played back, the magnetized segments produce electronic pulses representing binary values of one and zero.*

direct addressing; 2-23: *The location of data for an instruction is fully specified in the byte(s) that follows the instruction code.*

disk crash; 5-44: *Specifically refers to a disk read/write head making destructive contact with the surface of a rotating disk. Loosely refers to any disk unit failure that results in a system malfunction.*

disk memory unit; 5-42, 5-43: *A bulk storage memory unit that utilizes the surface(s) of magnetic disk(s) rotating at high speed for the storage of information used by a computer. Data is transferred at high speed to/from the disk using read/write head(s) similar to the type used in a digital magnetic tape unit.*

diskette; 5-45: *A thin, flexible platter (floppy disk) coated with magnetic material used as the storage medium in a floppy disk unit.*

DMA - direct memory access; 5-14, 6-36: *Provision in a computer system for the address and data lines to/from the memory elements to be directly controlled by devices other than the CPU so that memory contents may be directly accessed by those devices.*

dot matrix CRT display; 5-12: *A display that operates by illuminating selected dots on a grid of points.*

drum memory unit; 5-41: *A bulk storage memory device that utilizes a drum coated with magnetic material rotating at high speed for the storage of information. Data is transferred to/from the surface of the drum by read/write heads along paths or "tracks" defined as the area that passes directly underneath the heads as the drum spins.*

dynamic memory elements; 6-40: *Semiconductor memory elements that are not able to retain data more than a few thousandths of a second unless "refreshed" by activity on their address lines.*

Editor program; 3-53: *A program that allows the manipulation of text for editing purposes while it is stored in the memory banks of a computer. It is especially useful in the development of source listings of computer programs.*

effective address; 2-18: *The actual address where data is located. The final address obtained or calculated as a result of performing relative, indexed, indirect, or similar addressing operations.*

electro-mechanical printer; 5-10: *A machine capable of printing characters on paper by converting electrical signals into mechanical motion (through motors, cams, levers, linkages, and so forth).*

EPROM - eraseable programmable read-only-memory; 6-17: *A PROM device that may be erased (usually by exposure to intense ultraviolet light) and reprogrammed.*

even parity; 3-28: *Condition defined when the number of bit positions in a register set to the one state is an even*

value. (Note: the quantity zero is generally regarded as an "even" value for parity purposes.)

exchange instruction; 2-8: *An instruction to replace the contents of one register (or set of registers) with the contents of another and vice versa.*

EXCLUSIVE OR; 2-42: *A Boolean logic operation performed between specified registers on a bit-by-bit basis. A logic one is left in the results register if either bit in any bit position of either register is a one. A logic zero is left if both are a logic one or both are a logic zero.*

execute cycle; 2-4: *The portion of a computer's operational sequence when the actions required to perform an instruction are actually accomplished. Occurs after a "fetch" cycle has obtained an instruction from memory.*

Executive program; 4-6: *A program or portion of a program that essentially oversees or controls the overall operation of a multi-faceted system or program.*

extended addressing; 2-51: *Essentially the same as "direct" addressing in that the location of data for an instruction is fully specified in bytes that follow the instruction code. The "extended" implies that the address requires two bytes (whereas some direct modes may only require one byte).*

fan-in; 6-23: *Refers to the number of unit loads from which a particular circuit or device can accept inputs.*

fan-out; 6-23: *Refers to the number of unit loads a particular circuit or device can properly service or support (drive).*

fetch cycle; 2-4: *The portion of a computer's operational sequence when an instruction or portion of a multi-byte directive is obtained from the location in memory pointed to by the contents of the program counter.*

file gap; 5-34: *Short lengths of tape containing no recorded information used to separate "files" or "blocks" of data on a recording medium such as magnetic tape.*

files; 5-30, 5-34: *Refers to essentially arbitrarily-defined "blocks" of information making up a "record" or group of related data. Carries much the same connotation in computer work as it does when related to routine office work. A "file" is a "package" of information.*

firmware; 6-30: *A program supplied on a read-only-memory device. Refers to software that has effectively been made permanent by storing it in a piece of hardware!*

fixed head disk system; 5-44: *A disk unit in which the read/write heads are essentially fixed in position. Hence, one head is required for each track of information recorded on a disk.*

flag status register; 1-33: *A register used to store the states of the various flag cells. Each flag type is assigned a particular bit position within the register.*

flags; 1-33: *One-bit cells that serve to indicate whether or not certain conditions exist after various types of operations have been performed by a CPU.*

floppy disk unit; 5-45, 5-46: *A bulk storage memory device that uses a flexible platter coated with magnetic*

material as a storage medium. The unit mimics the operation of a moving head disk unit at a slower rate and reduced storage capacity.

flow charts; 3-4: *Special diagrams that illustrate the operation or flow of a computer program through the use of simple symbols such as circles, rectangles, and diamonds to denote key classes of operations. They are especially valuable for illustrating the relationships between various routines in a program.*

front panel; 6-21: *A panel equipped with switches and indicators for allowing an operator to manually control the operation of a computer system.*

functions; 4-24: *Supporting mathematical capabilities generally provided in high level language interpreters and compilers. Typically perform such mathematical operations as solving for the square root of a number, obtaining integer values, generating random numbers, and so forth.*

halt instruction; 2-59: *An instruction that directs a computer to stop program execution. Encountering such a directive typically results in a CPU remaining in a stopped state until an operator intervenes using manual controls to resume programmed operation.*

halt mode; 6-21: *A mode in which the computer ceases executing all instructions and remains in an idle condition.*

hard copy; 5-8: *Refers to copy printed on paper that may be kept as a permanent record versus information displayed on a temporary medium such as the screen of a CRT tube.*

hard error rate; 5-37: *Refers to the bit-error-rate attributable to permanent error conditions (when storing or recovering data to/from a bulk storage unit), such as those caused by defective equipment, defective transmission techniques, or faulty recording mediums.*

hardware; 5-19, 6-27: *Refers to the physical components or elements of a computer system.*

hexadecimal to binary conversion; 3-13: *Translation of a number written in hexadecimal format to binary format. Experience allows a person to expand a hexadecimal digit to a corresponding group of four binary digits once the appropriate patterns have been memorized.*

hexadecimal to decimal conversion; 3-17: *Translation of a number written using hexadecimal (base 16) notation to a number using decimal notation.*

hexadecimal numbering system; 3-13: *A numbering system having a base of 16 containing the digits 0 through 9, and the symbolic digits A through F. Frequently used by microcomputer programmers because hexadecimal digits are readily expanded to groups of four binary digits. Or, the condition of a group of four binary digits can be concisely recorded using one hexadecimal digit.*

high level language; 4-1: *A programming dialect that allows a user to program a computer at a much more convenient and efficient (from a programmer's viewpoint) level than machine language programming. A high level language generally utilizes a syntax structure that enables a computer to convert simply stated directives into*

appropriate sequences of operations which may be quite complex when actually performed at the machine language level.

high level programming; 3-1, 4-1: *Programming the operation of a computer by using a high level language such as BASIC instead of at the complex machine language level.*

high level source code; 4-5: *The statement lines (original directives) that a programmer prepares to direct the operation of a computer when using a high level language.*

I/O - input/output; 1-29, 5-1: *The passing of information into or out of a computer or reference to that process.*

I/O instructions; 2-55: *An instruction that results in information being inputted or outputted to/from a CPU.*

I/O interfaces; 6-19: *Circuit modules that adapt the signal requirements of external equipment to those of a computer and vice versa.*

I/O ports; 1-29, 5-3: *The physical arrangements (connection points) provided in a computer system so that data may be passed to/from the computer and external devices.*

IC - integrated circuit; 6-3: *A microminiature circuit fashioned using microphotography and mounted inside a special package. The package in which it is mounted typically has pins along its sides which allow the package to be plugged into a socket or soldered to a printed circuit card.*

immediate addressing; 2-16: *Data for an instruction is contained in the byte that immediately follows the instruction code.*

increment and skip if zero instruction; 2-59: *An instruction that increments the value of a register. If the value of the register goes to zero as a result of the increment operation, the next instruction in the current sequence will be skipped over. Otherwise, the next instruction will be performed.*

increment instruction; 2-58: *An instruction that causes the value of a register to be incremented (increased) in value, generally by a count of one. However, some machines provide capability for the increment size to be specified.*

indexed addressing; 2-20: *Data for an instruction is pointed to by the contents of an "index register" plus an offset value. The offset value is in the byte immediately following the instruction code.*

indirect addressing; 2-30: *Data for an instruction is stored at an address contained in a memory location indicated by some other addressing mode. Thus, the final address must be obtained indirectly.*

inherent addressing; 2-15: *Addressing information is contained within the instruction code as a function of the type of instruction.*

input device; 2-56: *Refers to any device which is capable of inputting information to a computer.*

input port; 2-67: *The physical arrangements (wires, electronic circuitry) that are provided in a computer system so that data may be passed into the computer from input devices.*

input signal line; 2-56: *A wire or electronic pathway used to pass information to a computer from an external device.*

instruction; 1-31: *At the machine language (most fundamental) level, it is a binary pattern that when decoded by the CPU will result in a specific action being performed by the computer. At other levels (such as a high level language program), it refers to any information that the computer takes as a directive.*

interface; 5-11: *Circuitry that buffers, adjusts for differences in voltage levels, provides appropriate control and logic signals, or performs whatever functions are required in order to allow two different devices to be electronically coupled to one another. Frequently required to permit various kinds of I/O devices to be connected to a computer.*

Interpreter; 4-6, 4-7: *A program that operates by immediately "interpreting" and executing statements as they are encountered in a high level language program.*

interrupts; 6-36: *The process of temporarily halting the execution of a main program in order to handle a secondary task at the signalling of an external device.*

invert; 2-42: *To reverse a binary signal state from a high to a low or from a logic '0' to a logic '1' condition.*

jump instruction; 2-46: *An instruction that directs a computer to obtain the next directive from a new location in memory rather than simply executing the next directive in the next successive location of memory. It effectively causes the program counter to be loaded with a new value rather than simply being incremented.*

keyboard; 5-8: *Refers to a typewriter-like input device that is used to input alphanumeric information to a computer in the form of appropriately coded binary patterns.*

keypad; 5-7: *A small keyboard containing relatively few keys, such as is commonly provided on simple calculator units. Typically used in small microcomputer systems as a simple, low cost, input device.*

keyword; 4-4, 4-16: *The word in a high level language statement line that defines the primary type of operation to be performed.*

labels; 3-22, 3-41: *Symbolic names used to represent addresses in memory or locations in a program.*

left shift instruction; 2-43: *An operation that shifts the contents of all bit positions in a register to the left. The MSB value is shifted out of the register. The LSB portion is commonly set to a value of zero.*

line drivers; 6-24: *Often has the same meaning as "buss drivers." Sometimes carries connotation of being designed to support (drive) physically long lengths of wire such as may be necessary when coupling a remote I/O device to a computer.*

load instruction; 2-7: *An instruction that transfers data from a memory location to a CPU register, or from one CPU register to another.*

logical AND instruction; 2-41: *A Boolean logic operation performed*

between specified registers on a bit-by-bit basis. A logic one is left in the results register whenever bits at the same positions in both registers are a logic one.

logical instructions; 2-42: *Refers to essentially any type of instruction that involves Boolean algebra operations.*

logical OR instructions; 2-41: *A Boolean logic operation performed between specified registers on a bit-by-bit basis. A logic one is left in the results register if any bit in any bit position of either register is a logic one.*

LSI - large scale integration; 6-1: *Refers to the process of manufacturing large numbers of microminiature electronic components on very small pieces ("chips") of semiconductor material.*

machine language programming; 3-2: *Directing the detailed operation of a computer at the most fundamental level by arranging the proper sequences of binary patterns representing instructions at specific addresses in a computer's memory banks.*

mainframe; 6-47: *Nomenclature that refers to the computer proper itself - consisting essentially of the CPU and memory elements - while excluding external parts of a system such as I/O devices.*

manual control module; 6-21: *A module that provides the same type of functional capabilities as a front panel.*

mass storage device; 5-25: *A device capable of saving large amounts of information from the memory banks of a computer for long term "inactive" storage.*

memory; 1-6, 6-10: *Elements in a computer that are able to retain binary coded patterns representing instructions or data which are utilized by the CPU.*

memory map; 3-22: *A memory usage or allocation diagram.*

memory pointer registers; 1-31: *Special registers that direct (point) the CPU to the location of a word in memory that contains data.*

microcomputer; 1-1: *A small computer system based on the use of a "CPU-on-a-chip."*

microprocessor; 2-1: *Refers to a "CPU-on-a-chip" or a computer system designed around such a device.*

mnemonics; 3-9: *A group of letters or alphanumeric characters grouped so as to assist a programmer in remembering a word or phrase. They are frequently created to serve as a shorthand notation for representing the various types of instructions and operations that a computer can perform.*

Monitor program; 3-56: *A program that enables an operator to conveniently direct a computer to perform frequently required tasks, such as loading or storing programs, examining or altering the contents of locations in memory, executing programs under controlled conditions, etc.*

moving head disk system; 5-44: *A disk unit in which a read/write head is capable of moving across the surface of the disk to access any one of a number of circular tracks of data.*

MSB - most significant bit; 1-34: *The left-most bit position in a multi-bit*

register. The cell position having the highest weighted value.

multiple-precision arithmetic; 2-33: *Arithmetic operations performed upon numbers of such magnitude that they must be stored in two or more registers.*

NOP - no-operation instruction; 2-60: *An instruction that simply consumes time when executed. It causes a computer to go through "fetch" and "execute" cycles, but otherwise does nothing.*

octal numbering system; 3-12: *A numbering system having a base of 8 containing the digits 0 through 7. Frequently used by microcomputer programmers because of its easy to learn relationships with the binary numbering system.*

octal to binary conversion; 3-12: *Translation of a number written in octal format to binary format. Expansion of an octal digit to a group of three binary digits can be done mentally by memorizing eight three digit binary patterns.*

octal to decimal conversion; 3-16: *Translation of a number written using octal (base 8) notation to decimal notation.*

odd parity; 3-29: *Condition defined when the number of bit positions in a register set to the logic one state is an odd value.*

operator precedence; 4-17: *The mathematical heirarchy or priority that an operator has in relation to other possible operators in a particular syntax structure (system of rules). The precedence determines the order in which operations will be performed when a mathematical expression is evaluated (solved).*

output device; 2-56: *Refers to any device which is capable of being operated by or accepting information from a computer.*

output port; 2-58: *The physical arrangement (wires, electronic circuitry) that is provided in a computer system so that data may be passed from the computer to output devices.*

paged machine; 2-26: *A computer that divides memory addresses into "blocks" of words referred to as "pages." Special directives are necessary in order to reach locations outside the current page pointed to by the program counter because the program counter is not provided with the capability to increment itself across a page boundary. It must be specifically set to a new page value.*

paging; 1-20, 2-24: *A convention of dividing address values in a computer into two parts - the high portion is referred to as a "page" value; the low part as the "location on a page." In microcomputers having eight-bit-wide memory words, a "page" is frequently defined as having 256 locations - the number of different addresses that may be specified in an 8-bit register. However, other page sizes are sometimes specified.*

paper tape punch; 5-28, 5-29: *An electro-mechanical machine capable of punching small holes in strips of paper tape material. The holes are arranged in patterns, and can represent binary coded information such as instructions or data used in a computer.*

paper tape reader; 5-24, 5-25, 5-28: *A device capable of detecting the presence or absence of holes punched in paper tape, and converting the patterns detected into electronic signals that can be passed on to a computer.*

parallel transmission; 5-5: *The transfer of information along a number of pathways at the same instant in time.*

parity flag; 1-36: *An indicator that signals whether or not the number of digits in the logic one condition is an odd or even value. The indicator is often used for error checking purposes during I/O operations.*

peripheral; 6-26: *Refers to any external or remote device in a computer system. I/O devices, for instance, are commonly referred to as "peripheral" devices.*

pop; 1-32: *The process of removing information from the "top" of a "stack," and then preparing to be able to access the next piece of information stored on the stack. This process generally involves changing the value of a "stack pointer" register so that it points to the next piece of data on the "pile" of data stored in memory.*

power supply, 6-24: *A unit that supplies the necessary voltages to a computer system in order to provide electrical power.*

program counter; 1-18: *A CPU register that holds the address value of the memory location where the next CPU directive is to be obtained.*

program development worksheet; 3-20: A paper form used to assist in the organized development of a complex computer program. Columns for *recording various types of pertinent information are provided. They are filled in as a programmer works through successive stages in the program development and assembly process.*

PROM - programmable read-only-memory; 6-16: *A type of memory element similar to a ROM except that it may be initially programmed with a desired set of patterns (using suitable equipment) by an individual rather than at the factory. Once programmed, the patterns are permanent as in a standard ROM device.*

pseudo-paging; 2-26: *A way of viewing memory locations whereby programmers refer to memory addresses as being organized into "blocks" of words referred to as "pages" for reference purposes only. The machine being used may not be a true paged machine in that the program counter is able to routinely advance from the end of one block of addresses to the start of the next group - an operation that cannot be performed by a true paged machine.*

punched paper tape; 5-26: *Paper tape that has had rows of holes punched in selected columns so as to represent binary information that may be used in a computer system.*

push; 1-32: *The process of placing information on the "top" of a "stack" and then preparing to be able to place the next piece of information on the stack. This process generally involves changing the value of a "stack pointer" register so that it points to the next address where data may be stored.*

RAM - random-access-memory; 1-9, 5-22, 6-11, 6-39: *Memory elements*

arranged such that any location may be accessed as easily as any other without regards to the address of the last elements accessed. The term is also popularly used to denote semiconductor "read and write" memory elements that are volatile (subject to losing their contents upon loss of power).

refreshing; 5-14, 6-40: *Refers to the process of constantly reactivating or restoring information that decays or fades away when left idle. For instance, the phosphor on a CRT screen needs to be constantly reactivated by an electron beam in order to remain illuminated. Cells in dynamic memory elements must be repeatedly accessed in order to avoid a fading away of their contents.*

register; 1-25: *A collection of cells which is referred to for whatever functional purposes as an operational group.*

register indirect addressing; 2-16: *An addressing mode that utilizes the contents of "memory pointer" registers to indicate a memory address.*

relative addressing; 2-16: *Data for an instruction is contained in a memory location that is positioned a specified number of words away relative to the current contents of the program counter. The relative offset value (to be added to the program counter value in order to reach the location) is supplied in the byte that follows the instruction code.*

return instruction; 2-51: *An instruction that terminates a "subroutine" causing program execution to "return" to the point immediately following the "call" class of instruction that initiated entry to the subroutine.*

right shift instruction; 2-43: *An operation that shifts the contents of all bit positions in a register to the right. The least significant bit value is shifted out of the register. The MSB portion is typically set to a value of zero.*

ROM - read-only-memory; 1-9, 6-15: *Refers to memory devices that contain fixed patterns that cannot be altered. The patterns on these devices are not affected by loss of power. Hence, they are valuable for holding programs that make a computer system operational as soon as power is applied.*

rotate left instruction; 2-44: *An operation similar to a left shift operation except that instead of information being shifted out of a register from the MSB position, the digit in that position is brought around to the other end of the register to occupy the LSB position.*

rotate right instruction; 2-44: *An operation similar to a right shift operation except that instead of information being shifted out of a register from the LSB position, the digit in that position is brought around to the other end of the register to occupy the MSB position.*

rotating a register; 2-44: *An operation similar to a shift except that instead of information being shifted completely out of a register, it is brought around to the end of the register where a bit position would be vacated by the shifting operation.*

routine; 3-3: *A sequence of instructions. Generally the sequence referred to is considered to have some logical*

relationship with other instructions in the sequence or be capable of performing some specific operation as a group. A logical segment of a program. An algorithm.

run mode; 6-21: *A mode in which the computer is considered to be operating when it is automatically executing instructions contained in its memory banks.*

run time routines; 4-6, 4-7: *Sequences of instructions capable of performing specific operations which are used by programs that have been compiled by a high level language compiler. They are used when it is time to execute ("run") the machine language programs that have been produced.*

scrolling; 5-18: *Refers to the process of removing a line of information from a temporary display such as a CRT, then moving all other lines being displayed up or down so as to permit a new line of information to be displayed in the area vacated by the line that has been removed. The effect to a person viewing the process is similar to what one would observe when viewing sections of information stored on scrolls.*

semiconductor; 6-1: *Refers to material chemically treated to have electronic characteristics which lie between those of a true metallic conductor and those of an insulator. A material frequently used consists of silicon mixed with minute quantities of other elements to alter its normal electrical characteristics.*

serial transmission; 5-4: *The transfer of information over a single pathway by changing the signal condition as a function of time.*

shift instructions; 2-43: *An operation that shifts the values of all bit positions in a register to the right or left. This type of operation results in data being shifted completely out of one end of a register (depending on the direction of the shifting operation).*

sign flag; 1-34: *An indicator that is set to a logic one if the MSB of a register being tested is a binary '1.' When using the "two's complement" arithmetic convention, this condition indicates that the register contains a negative number.*

single-step mode; 6-21: *A mode that allows an operator to have the computer execute just one instruction at a time. The operator can thus closely examine the results of each operation to determine if an algorithm performs as planned.*

soft error rate; 5-37: *Refers to the bit-error-rate attributable to transient conditions in a bulk storage system. Such errors may theoretically be eliminated by repeating the transfer operation.*

software; 6-22: *Refers to any program, algorithm, or routine for a computer.*

source code; 3-53: *Refers to the original mnemonic or high level statement versions of a program. The starting information or "source" from which the final "object" (machine language or final code) is derived.*

stack pointer; 1-32, 2-52, 6-41: *A special register that enables a CPU to process "stacks" ("piles" or "nests") of data using a first in - last out (push - pop) or similar method. The stack pointer register "points" to (contains the address of) the current "top" or entry point of a stack of information.*

statements; 4-4, 4-16: *High level language expressions that define types of operations to be performed by the computer. A series of appropriate statements forms a high level language program.*

static memory elements; 6-39: *Semiconductor memory devices that retain data indefinitely without any further attention as long as power is supplied.*

stepping motors; 5-29: *A motor that instead of spinning or rotating at a constant speed, operates by rotating its shaft a specific number of degrees each time it is activated. If activated rapidly (pulsed), such a unit appears to behave like a regular motor. It has an attractive feature for certain applications in that it can be stopped almost instantaneously by stopping the input of its controlling pulses.*

store instruction; 2-7: *An instruction that transfers data from a CPU register to a location in memory.*

subroutine; 2-50, 4-20: *A routine or program segment that may be used many times by different sections of a program. It is terminated by a "return" directive which causes program execution to revert back to the location from where a "call" instruction initiated usage of the routine.*

subtract instruction; 2-36: *A mathematical subtraction operation provided as a convenience instruction in most microprocessors. The operation is generally performed by invoking a two's complement addition procedure.*

symbolic names; 3-22, 3-41: *Names assigned by programmers to represent addresses or locations in a program. (Also sometimes refers to names given to variable values in high level language programs.)*

teletypewriter; 5-19: *Refers to an electro-mechanical machine consisting of a typewriter-like keyboard and printer. Such machines were originally used to transmit information in communications systems. They are now frequently used to communicate with a computer. Many of the machines are designed to utilize ASCII or other standardized codes.*

terminal; 5-18: *Refers to essentially any kind of device that may be used to communicate with a computer. The terminology is perhaps more often used in the context of referring to devices having alphanumeric capability.*

TV - television display; 5-8: *When associated with microcomputer systems, the nomenclature refers to an interface or module that can display alphanumeric and/or graphic information generated by a computer on an ordinary television set.*

two's complement arithmetic; 2-36: *A convention whereby all positive binary numbers are denoted by having a zero digit in the MSB of a register. Negative numbers are handled as the "complement value plus one" of positive numbers. Hence, they are readily identified as having a one in the MSB position of a register. A positive binary number and it's two's complement value will add up to a value of zero!*

unblanked; 5-14: *Refers to Z-axis or intensity modulation of the display beam in an oscilloscope or CRT. When the screen is unblanked, it will be illuminated because electrons are able to strike the phosphor coated screen.*

variable name; 4-18: *A symbolic representation of a variable (or unknown) value in a mathematical expression. Commonly used when programming in a high level language.*

vector generating CRT display; 5-13: *A display that operates by generating lines (vectors) between selected points on a screen.*

virtual copy; 5-23: *Producing an essentially exact copy of the contents of locations in the memory of a computer onto an external bulk storage device - including reproducing the information regarding the exact memory addresses in which the data being stored resides. The addressing information is required so that the data can later be restored to the exact same locations in the exact same format in the computer as though the memory contents had never been altered. The process of making a "virtual copy" allows a program to be completely removed and then later restored to a computer leaving no trace of the interruption!*

word; 1-16: *A group, row, or register of binary cells that is typically processed in one simultaneous operation. Generally, the word size in a microcomputer is defined as the number of binary cells contained in the system's accumulator(s).*

worst case access time; 5-38: *The maximum amount of time required to go between the two most widely separated points (addresses) on a bulk storage device. For a tape unit, this would be the time required to traverse the entire length of the tape. For a fixed-head disk unit, it would be the time required for the disk to make one revolution.*

zero flag; 1-33: *An indicator that is set to a logic one condition if a register being tested contains all zeros in its cell positions. It is set to a logic zero state if any cell in the register is in a '1' condition. It is simply used to indicate whether or not a register is empty!*

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

For a brochure describing current SCELBI publications, write to:

SCELBI COMPUTER CONSULTING, INC.
P. O. Box 133 - PP STN
Milford, CT 06460